

Advanced Revelation Version 2.0

RevelationTechnologies

October 1989



COPYRIGHT

Copyright © 1989 by Revelation Technologies, Incorporated
All Rights Reserved

No part of this publication may be reproduced by any means, be it transmitted, transcribed, photocopied, stored in a retrieval system, or translated into any language in any form, without the written permission of Revelation Technologies, Inc.

Printed in the United States of America

TRADEMARK NOTICE

Revelation, Advanced Revelation, and Environmental Bonding are trademarks of Revelation Technologies, Inc.

dBASE and dBASE III are trademarks of Ashton-Tate.

Microsoft, MS-DOS, and OS/2 are trademarks of Microsoft Corporation.

ABOVE BOARD and Intel are trademarks of Intel Corporation.

NetWare and Advanced NetWare are registered trademarks of Novell, Inc.

Lotus and Lotus 1-2-3 are trademarks of Lotus Development Corporation.

Quarterdeck Expanded Memory Manager is a trademark of Quarterdeck Office Systems.

Rampage is a trademark of AST Research, Inc.

XMA2EMS is a trademark of International Business Machines, Inc.

386^{MAX} is a trademark of Qualitas, Inc.

SOFTWARE COPYRIGHT NOTICE

Your license agreement with Revelation Technologies, Inc., authorizes the conditions under which copies of the software can be made and the restrictions imposed on the computer system(s) on which they may be used. Any unauthorized duplication or use of any software product produced by Revelation Technologies, Inc., including Advanced Revelation, in whole or in part, in any manner, be it in print or in any other storage-and-retrieval system, is forbidden.

Advanced Revelation, Version 2.0
ISBN 0-923387-33-1
October 1989



Table of Contents

Volume I

Chapter A 1

Introduction to Advanced Revelation

About Advanced Revelation	A 1.1
About the Documentation	A 1.3
Conventions Used in this Book	A 1.4

Chapter B 1

Installing Advanced Revelation

Preparing Your System for the Installation	B 1.1
Overview	B 1.1
Installing Advanced Revelation	B 1.2
Installing Optional Modules	B 1.4

Chapter B 2

Updating and Enhancing Your Advanced Revelation System

About Advanced Revelation Installation Utilities	B 2.1
Adding Optional Modules to Your System	B 2.1
Updating or Upgrading Advanced Revelation	B 2.3
Using the Installation Utility	B 2.4
Upgrading User Accounts	B 2.6
Note to the System Administrator	B 2.9

Chapter B 3

Removing Optional Modules

Removing Optional Modules	B 3.1
-------------------------------------	-------

Chapter B 4

Logging On to Advanced Revelation

Logging On to Advanced Revelation	B 4.1
---	-------

Chapter B 5

Using Expanded Memory Support

Introduction	B 5.1
System Requirements	B 5.1
Using Expanded Memory Logon Options	B 5.2
Managing Expanded Memory	B 5.4
Displaying the Status of Expanded Memory	B 5.5
System Configuration	B 5.6

Chapter C 1

Introduction to the Online Tutorial

Introduction to the Online Tutorial	C 1.1
Accessing the Tutorial	C 1.1
Running a Tutorial Lesson	C 1.5

Chapter D1

Getting Started

Starting, Moving Around, and Leaving	D 1.1
Getting Online Help	D 1.4
Other Important Keys	D 1.5

Chapter D 2

Using System Features

The Advanced Revelation Environment	D 2.1
Managing the Advanced Revelation Environment	D 2.2
The Online Help System	D 2.2
What the Status Line Tells You	D 2.4
Using Menus	D 2.6
Using Windows	D 2.7
Using Popups	D 2.18

Chapter D 3

Entering and Editing Data in Windows

Moving the Cursor	D 3.1
Entering and Editing Data in Windows	D 3.6

Chapter D4

Using the Full-Screen Editor

Introduction to the Full-Screen Editor	D 4.1
Using Defined Blocks of Text	D 4.5
Searching and Replacing	D 4.8
Using Value Windows	D 4.9
Using Softkeys in the Full-Screen Editor	D 4.10

Chapter E 1

Advanced Revelation Files and Records

About Files & Records	E 1.1
Terminology: Files, Records, and Fields	E 1.1
Characteristics of Advanced Revelation Files	E 1.2
Characteristics of Advanced Revelation Records	E 1.5
Characteristics of Advanced Revelation Fields	E 1.6

Chapter E 2

Managing Files and Records

Defining Files and Fields	E 2 . 1
Defining a File Using the Definefile Window	E 2 . 1
Advanced Revelation File Attributes	E 2 . 13
Creating File Synonyms	E 2 . 15
Adding or Modifying Advanced Revelation Commands (the VOC File)	E 2 . 16
Transaction Processing	E 2 . 18
Advanced Revelation System Files	E 2 . 28

Chapter E 3

Advanced Revelation Dictionaries

About Advanced Revelation's Dictionaries	E 3 . 1
Creating a Data Dictionary	E 3 . 1
Dictionary Record Types	E 3 . 3
Creating Temporary Fields for Reports	E 3 . 6
Paint Autobinding	E 3 . 7
Available Dictionary Reports	E 3 . 7
Dictionary Characteristics and Record Format	E 3 . 8

Chapter E 4

Advanced Revelation Volumes

About Volumes	E 4 . 1
Attaching Files in a Volume	E 4 . 2
Creating a Volume	E 4 . 4
Additional Volume Functions	E 4 . 5

Chapter F 1

Building Applications with Advanced Revelation

Introducing the Application Development Process	F 1 . 1
---	---------

Chapter F 2

Planning an Application

Planning Checklist	F 2 . 1
Establishing Application Requirements	F 2 . 1
Developing an Application Design	F 2 . 3
Designing the Advanced Revelation Structure	F 2 . 9
An Application Planning Worksheet	F 2 . 13

Chapter F 3

Users & Accounts: Planning Access and Security

Introducing Users and Accounts	F 3 . 1
Planning for Security	F 3 . 4
Setting Up Accounts	F 3 . 9
Setting Up Users	F 3 . 11
Setting Up Environments	F 3 . 13
Setting Video Attributes	F 3 . 15
Assigning Environment and Video Characteristics	F 3 . 17

Chapter F 4

Creating Windows with Paint

Overview of the Painting Process	F 4 . 1
Getting Started	F 4 . 2
Adding, Changing, and Removing Prompts and Entry Lines	F 4 . 8
Adding, Changing, and Removing Labels	F 4 . 21
Using Paint Options	F 4 . 22
Resizing and Moving Your Window	F 4 . 26
Testing Your Window with Test Run	F 4 . 29

Chapter F 5

Adding Advanced Features to Windows with Paint

The Advanced Paint Options	F 5 . 1
Grouping and Reordering Prompts	F 5 . 2
Using Default and Duplicate Prompts	F 5 . 3
Defining Advanced Prompt Characteristics	F 5 . 4
Defining Smart Labels	F 5 . 11
Using Prompt Formulas	F 5 . 11
Displaying, Editing, and Updating Dictionaries	F 5 . 13
Data Validation	F 5 . 19
Data Formatting	F 5 . 27
Quick Painting Windows	F 5 . 32
Using the Paint Indexing Menu	F 5 . 33
Customizing Windows	F 5 . 33
Creating Custom Softkeys	F 5 . 39
Relating Windows	F 5 . 39
Setting Window Video Attributes	F 5 . 41
Establishing Window Security	F 5 . 41
Disabling Window Keys	F 5 . 43

Chapter F 6

Building Menus and Popups

Designing and Building Menus	F 6 . 1
Displaying a Main Menu in an Application	F 6 . 3
Designing and Building Popups	F 6 . 4
Displaying Popups with Codes and Commands	F 6 . 13

Chapter F 7

Creating and Maintaining Indexes

Introduction	F 7 . 1
Creating Indexes	F 7 . 3
Using Quickdex	F 7 . 10
Indexing System Maintenance	F 7 . 13
Reviewing Index Specifications	F 7 . 16
Updating and Rebuilding Indexes	F 7 . 17
Developer Notes for Indexing	F 7 . 22

Chapter F 8

Building Formulas

What is an R/BASIC Formula?	F 8 . 1
Accessing the Formula Window	F 8 . 1
Glossary of R/BASIC Formula Trigger Words	F 8 . 4

Chapter F 9

Using Codes and Commands

Where to Use Codes and Commands	F 9 . 1
The Codes and Commands	F 9 . 5
The Code and Command Options	F 9 . 12

Chapter F 10

Using the Batch Update Process

About Batch Updating	F 10 . 1
Creating a Batch Update Process	F 10 . 1

Chapter F 11

Writing Macros and Capturing Keystrokes

Defining and Using Macro Sets	F 11 . 1
Capturing Keystrokes	F 11 . 3

Chapter G 1

Selecting and Browsing Through Data

Using Record Keys	G 1.1
Defining Selection Criteria for Record Key Lists	G 1.3
Using Browse Lists in Application Windows	G 1.4

Chapter G 2

Using the Filter Key

Using the Filter Key	G 2.1
--------------------------------	-------

Chapter G 3

Using Indexes

Introducing Advanced Revelation's Indexing System	G 3.1
Index Search Options in Application Windows	G 3.1

Chapter G 4

Using Query

Using Query in Application Windows	G 4.1
Using Query from TCL	G 4.9
Modifying the Environment for Query	G 4.11

Chapter H 1

Creating Reports with Advanced Revelation

Overview of Advanced Revelation Reports	H 1.1
Overview of Advanced Revelation Report Tools	H 1.2
Customizing the Data Dictionary for Reports	H 1.2
Using View for R/LIST Reports	H 1.4
Developer Notes for View	H 1.6

Chapter H 2

Creating Forms

Creating Forms	H 2.1
Specifying Form Characteristics	H 2.7
Testing a Form	H 2.10
Printing Forms	H 2.10

Chapter H 3

Creating Labels

Creating Labels	H 3.1
Testing a Label Template	H 3.3
Printing Labels	H 3.3

Chapter H 4**Creating R/LIST Reports**

Creating R/LIST Reports	H 4 . 1
Using the Make an RList Report Window	H 4 . 1

Chapter H 5**Writing Reports with EasyWriter**

Introducing EasyWriter	H 5 . 1
Components of EasyWriter	H 5 . 5
Developer Notes for EasyWriter	H 5 . 14

Chapter H 6**Creating Merge Reports**

Introducing Advanced Revelation's Merge Processor	H 6 . 1
Creating a Merge Template	H 6 . 2
Entering Scripts	H 6 . 7
Executing a Merge	H 6 . 19
Special Features of the Merge Processor	H 6 . 21

Volume II**Chapter I 1****Introduction to TCL**

Introducing TCL	I 1 . 1
---------------------------	---------

Chapter I 2**Features of TCL**

Features of TCL	I 2 . 1
---------------------------	---------

Chapter I 3**TCL Command Reference**

TCL Command Reference	I 3 . 1
---------------------------------	---------

Chapter J 1**Introduction to R/LIST**

Introduction to R/LIST	J 1 . 1
Using Filters	J 1 . 6

Chapter J 2**R/LIST Keyword Reference**

R/LIST Command Format	J 2.1
R/LIST Format Modifier Keywords	J 2.3
R/LIST Format Override Keywords	J 2.5
R/LIST Throwaway Words	J 2.6
R/LIST Keywords	J 2.6

Chapter K 1**The SQL Option**

Advanced Revelation SQL	K 1.1
Getting to SQL	K 1.1

Chapter K 2**An Introduction to SQL**

Introduction	K 2.1
The Database According to SQL	K 2.1
SQL and Advanced Revelation	K 2.7

Chapter K 3**Building a Database**

Creating Tables	K 3.1
Viewing Information	K 3.8

Chapter K 4**Querying the Database**

A Query Language	K 4.1
Forming Queries	K 4.1

Chapter K 5**Modifying the Database**

Inserting Rows	K 5.1
Updating Columns	K 5.3
Deleting Rows	K 5.5
Deleting Tables and Views	K 5.6

Chapter K 6**Managing the System**

The SQL Environment	K 6.1
Accessing the Dictionary	K 6.3
Maintaining Security	K 6.4
Controlling Transaction Processing	K 6.4

Chapter K 7

Programming with SQL

Introduction	K 7 . 1
The SQL Module Language	K 7 . 1
Embedded SQL	K 7 . 2

Chapter K 8

SQL Reference

Extensions to SQL	K 8 . 1
Reserved Words	K 8 . 4
Advanced Revelation SQL Data Types	K 8 . 5
SQL Operators	K 8 . 6
SQL Keywords	K 8 . 7
SQL Commands	K 8 . 32

Chapter L 1

Introduction to Networks

Introduction	L 1 . 1
Advanced Revelation and Networks	L 1 . 3

Chapter L 2

Managing Advanced Revelation on a Network

Managing Files on a Network	L 2 . 1
The Advanced Revelation Environment	L 2 . 2
Executing Network Commands	L 2 . 5
Managing Indexing on a Network	L 2 . 6
Other Network Management Issues	L 2 . 8

Chapter L 3

Locking

Introduction to Locking	L 3 . 1
Types of Locking	L 3 . 2
Establishing Locking	L 3 . 5
Setting Locking Options	L 3 . 7
Implicit Locking	L 3 . 8
Additional Lock Issues	L 3 . 13

Chapter M 1

Introduction to Environmental Bonding

Environmental Bonding	M 1 . 1
Tenets of Environmental Bonding	M 1 . 3
Using an Environmental Bond	M 1 . 4
Characteristics of Environmental Bonding	M 1 . 8

Chapter M 2**The ASCII Environmental Bond**

Description of the ASCII Bond	M 2 . 1
Using the ASCII Bond	M 2 . 3

Chapter M 3**The dBASE III Environmental Bond**

Description of the dBASE III Bond	M 3 . 1
Using the dBASE III Bond	M 3 . 3
The dBASE III Bond Reference	M 3 . 12

Volume III**Chapter N 1****Introduction to R/BASIC**

R/BASIC Overview	N 1 . 1
----------------------------	---------

Chapter N 2**Features of R/BASIC**

Features of R/BASIC	N 2 . 1
-------------------------------	---------

Chapter N 3**Programming with R/BASIC**

Programming with R/BASIC	N 3 . 1
------------------------------------	---------

Chapter N 4**R/BASIC Command Reference**

R/BASIC Command Reference	N 4 . 1
-------------------------------------	---------

Chapter N 5**R/BASIC System Subroutines**

R/BASIC System Subroutines	N 5 . 1
--------------------------------------	---------

Glossary**Index**

List of Tables

Volume I

Chapter D 2

Using System Features

Table D2-1 Kinds of Fields

Chapter D 3

Entering and Editing Data in Windows

Table D3-1 Determining Editor Status

Chapter D 4

Using the Full-Screen Editor

Table D4-1 Markers Used to Separate Field Values

Chapter E 1

Advanced Revelation Files and Records

Table E1-1 Advanced Revelation Data Types

Chapter F 3

Users & Accounts: Planning Access and Security

Table F3-1 How Passwords Can Restrict Access

Table F3-2 How Logon Affects Security

Table F3-3 How Logto Affects Security Restrictions

Table F3-4 Planning Window Security

Table F3-5 How Logon Affects Environment and Video Characteristics

Chapter F 4

Creating Windows with Paint

Table F4-1 Dictionary Types

Table F4-2 Prompt Entry Type Options

Chapter F 5

Adding Advanced Features to Windows with Paint

Table F5-1 Date Conversions

Table F5-2 Time Conversions

Table F5-3 Decimal Numbers and Money Conversions

Table F5-4 Conversion Mask Samples

Chapter F 7**Creating and Maintaining Indexes**

Table F7-1 Index Locations

Chapter G 3**Using Indexes**

Table G3-1 Index Search Methods

Chapter G 4**Using Query**

Table G4-1 Operators Available in Query

Table G4-2 Quick Reference Chart of Query Syntax

Chapter H 6**Creating Merge Reports**

Table H6-1 Format Flags for the Merge Processor

Table H6-2 Quick Reference Chart for the Merge Processor

Volume II**Chapter I 3****TCL Command Reference**

Table I3-1 Terminal Emulation Settings

Chapter K 2**An Introduction to SQL**

Table K2-1 Types of Advanced Revelation SQL Commands

Chapter K 7**Programming with SQL**

Table K7-1 Embedded SQL Commands

Chapter K 8**SQL Reference**

Table K8-1 SQL2 Proposed Extensions

Table K8-2 Advanced Revelation SQL Keyword Extensions

Table K8-3 Advanced Revelation SQL Reserved Words

Table K8-4 Advanced Revelation SQL Data Types

Table K8-5 SQL Operators

Chapter L3

Locking

Table L3-1 Lock Table

Chapter M 1

Introduction to Environmental Bonding

Table M1-1 Advanced Revelation Standard Data Types

Chapter M 3

The dBASE III Environmental Bond

Table M3-1 dBASE Bonding Dictionary

Table M3-2 dBASE to Advanced Revelation

Table M3-3 Advanced Revelation to dBASE

Volume III

Chapter N 3

Programming with R/BASIC

Table N3-1 Debugger Commands



Chapter A 1

Introduction to Advanced Revelation

In this chapter you are introduced to Advanced Revelation. Here you can read about the major features of Advanced Revelation and about why Advanced Revelation is the development environment of choice for today's PC applications. In addition, you will find information about this documentation set and about the conventions used in these books.

About Advanced Revelation	A 1 . 1
About the Documentation	A 1 . 3
Conventions Used in this Book	A 1 . 4



About Advanced Revelation

Welcome to Advanced Revelation! You are about to enter a new world of data management, one in which the only limitation on what you can accomplish is your imagination.

Managing Data with Advanced Revelation

Today's world of information management relies increasingly on the personal computer. Although data may be stored on everything from PCs to mainframe computers, it is the PC that links you as a computer user into the "enterprise-wide" computing that characterizes today's business.

Advanced Revelation is the perfect data management tool for today. Combining ease of use with power, Advanced Revelation's application development and management tools allow you to create and use everything from simple customers lists to "mission-critical" company-wide accounting and manufacturing systems.

In addition, Advanced Revelation includes facilities that allow you to access information anywhere in your system. Advanced Revelation's SQL provides an industry-standard language for data manipulation and management. And Advanced Revelation's unique Environmental Bonding capability allows you to access data in the format of almost any product or system, from spreadsheets and CAD drawings to database servers and mainframe computers.

In short, no matter what your information management requirements, Advanced Revelation is the environment of choice. As you work with Advanced Revelation and explore its possibilities, you will find that there is nothing that you will not be able to accomplish.

Features of Advanced Revelation

Advanced Revelation includes a complete array of data management tools. Listed below is a sampling of the tools and utilities that constitute Advanced Revelation's superior development environment.

Online Help In addition to this documentation set, Advanced Revelation features extensive online help. You can press keys almost anywhere in Advanced Revelation to learn more about the current process or to display options and their corresponding keystrokes.

File Management Featuring the most flexible file management system available today, Advanced Revelation puts virtually no limit on how large your files, records, and fields can be. Advanced Revelation's dictionary-driven database management system stores all information about a file in a central dictionary that can be changed at any time.

A powerful indexing facility allows you to speed your searches and sorts, while enabling you to look up data using any data in a record. A unique feature of Advanced Revelation's indexes is that they can be updated when your system is idle or by another workstation on a network.

Application Development

To create your applications, you can use the most innovative development tools available. For example, Advanced Revelation's Paint tool allows you to design and build an entry window and a file at the same time by simply painting prompts on the screen. You can develop menus, popups, and reports just as easily by filling in prompts in a window.

The application development tools are as powerful as they are simple to use. You can change any aspect of a window or menu by simply editing its "template". In addition, you can customize all aspects of an application extensively with little or no programming, saving you hours or days of development effort — even for the most complex applications.

Reporting

Using Advanced Revelation's reporting tools, you can create almost any type of report that your application will require. R/LIST provides you with columnar reports, while a forms processor allows you to "paint" reports on a screen. A label processor handles almost any type of mailing label, and an extremely powerful merge processor allows you to merge database information with pre-defined letters or other documents.

SQL

In addition to other Advanced Revelation data management tools, Advanced Revelation's SQL puts the power of an industry-standard language at your fingertips. Designed to be compliant with the ANSI standard, Advanced Revelation's SQL offers you powerful features in a language that is easy to learn as well as portable.

Networking

If you plan to run applications on a local area network, Advanced Revelation is ready with support for all major networks. Advanced Revelation also provides a wide range of options for coordinating multi-user access to data.

Environmental Bonding

One of the most innovative capabilities in any database manager, Environmental Bonding allows you full access to data stored in other formats or other locations. For example, your application can read and write data in dBASE III files, ASCII files, files on a database server, or in a host of other environments.

Programming For applications that require programming, Advanced Revelation makes available R/BASIC, the best database programming language available today. Featuring dozens of functions and commands especially suited to database programming, R/BASIC also includes a compiler, an interactive debugger, a full-screen editor, and other important capabilities.

Other Products Your data management needs don't stop with a single product or application. To assist you with all your needs, a full Advanced Revelation product line is available. Additional products include:

- Advanced Revelation Runtime systems that allow users to execute but not change an application.
- Network Bump Disks that turn your single-user copy of Advanced Revelation into a version capable of supporting multiple users on a network.
- Developer documentation and products, including in-depth information on programming with R/BASIC, filing systems, window processing, and more, as well as interface packages for C and assembly language programming.
- Technical Support, including phone assistance, and the Advantage for Developers program, a package of products and services for high-level developers.
- Education on Advanced Revelation from a worldwide network of Authorized Training Centers.

About the Documentation

The documentation for Advanced Revelation comes in a series of binders. The documentation set as a whole is divided into tabbed sections that are identified with a single letter. For example, the section called SQL is section K and is identified by a tab.

Within each section you will find one or more chapters. The chapters are identified with the section letter followed by the chapter number. To use the example of the SQL section again, the second chapter is identified as K2.

A table of contents appears at the front of the first volume of the documentation. There are also individual tables of contents for each section and for each chapter. A glossary and an index appear at the end of the last volume.

Here is a small sample to show you this organization:

Volume I

Section F (*Developing an Application*)

Chapter F1 ("Building Applications with Advanced Revelation")

Chapter F2 ("Planning an Application")

Chapter F3 ("Users and Accounts: Planning Access and Security")

etc.

Conventions Used in this Book

The following conventions are used in the documentation. Refer to the following as you read, especially if you are reading about TCL, SQL, or R/BASIC commands.

- ⇒ Continuation. Text is split in the book because of margins, but should be entered on one line in Advanced Revelation. Example:

```
SELECT COMPANY, ADDRESS, CITY FROM    ⇒
CUSTOMER_TABLE ORDER BY CITY
```

- { } Optional expression. The text or expression in the braces is an optional part of the command or expression. Example:

```
PAINT {filename} template.name
```

- [] Choice of expressions. Choose one from the expressions listed inside the elongated brackets. Example:

```
WRITE record [ON] filevar ...
              [TO]
```

- :: Repeat expression. Indicates that the expression or expressions can be repeated one or more times. Example:

```
CREATE VIEW view {column ::} AS subquery
{WITH CHECK OPTION}
```

```
CREATE TABLE [column type {DEFAULT value}
{column_def} ::] {ON vol}
```

- lower case** Variable information. The word in lower case indicates a position that will be filled with a real name or expression when you actually execute a command. (See also *Italics*.) Example:

```
LIST filename BY sort_criteria
```

- Boldface** Menu option. Boldfaced words refer to the literal names of items on an Advanced Revelation menu. Example:

Choose the option **Tools-Files-Indexing** ...

Italics Prompt names. Names of prompts in Advanced Revelation appear in italics. Italics are also used for variable information if the text around it is already in lower case. Examples:

At the *Volume name* prompt ...

or

The name *filename* is not valid ...

Courier Text Examples or messages. Text in courier indicates an example or a message that you might see in Advanced Revelation. Examples:

LISTDICT filename

or

Creating DICT.filename



Chapter B 1

Installing Advanced Revelation

Use this set of instructions when you are first loading Advanced Revelation onto your computer or network.

Preparing Your System for the Installation	B 1 . 1
Overview	B 1 . 1
Installing Advanced Revelation	B 1 . 2
Installing Optional Modules	B 1 . 4



Preparing Your System for the Installation

Before installing a full copy of Advanced Revelation make sure that your system is properly prepared.

Requirements

In order to install Advanced Revelation successfully, you must have the following:

- IBM PC, PC AT, PS/2, Compaq, or 100 percent compatible (for optimal processing speed a PC AT or faster machine is strongly recommended).
- At least 640K RAM. If you use RAM-resident utilities such as “popup” notepads or communications or network programs, you may wish to remove these before installing Advanced Revelation in order to free as much RAM as possible.
- At least 5MB disk space on a hard disk
- One 5-1/4" or one 3-1/2" drive
- Proper read/write rights to the target directory. This is particularly important if installing directly onto a network drive. Because Advanced Revelation creates its own subdirectories, be sure that you also have subdirectory creation (parental) rights.

For information on how to determine the amount of RAM and disk space you have available, or for information on your operating system or your network, consult the documentation which accompanies your computer or your network.

Overview

The initial installation of Advanced Revelation is a two part process:

- Install the base set of Advanced Revelation files.
- Install optional modules.

☛ Before attempting an Advanced Revelation installation, back up all the disks provided in the package. Use only backup copies of the disks when installing Advanced Revelation.

When you are ready to install Advanced Revelation, follow the steps in “Installing Advanced Revelation” to copy the base set of Advanced Revelation files to your hard disk.

After installing all of the files needed to run Advanced Revelation, the Installation program asks if you would like to install optional modules, such as the dBASE and ASCII bonds, that are provided with the base product. If you answer yes, the program logs you into Advanced Revelation and displays the Installation Utilities menu. You can use the Installation Utilities menu to install optional modules. Complete the steps in “Installing Optional Modules” in this chapter.

- Optional modules can be added at your convenience. If you do not want to install optional modules at this time, answer no when prompted. This stops the installation process. If you decide to add optional modules to an Advanced Revelation system that has been used for any length of time, follow the instruction in the “Updating and Enhancing your Advanced Revelation System” chapter.

Installing Advanced Revelation

Follow the steps listed below to install the base set of Advanced Revelation files on your hard disk. At any time, you can halt the installation process and return to the operating system by pressing [Esc] and selecting **QUIT** at the menu.

- To make a menu selection, use the arrow keys to highlight the option of your choice and press [Enter].

Step 1

Begin by starting up, or “booting” your computer so that the operating system prompt, in most cases C>, displays on the screen.

Step 2

Put the Advanced Revelation Install Disk into your computer’s diskette drive A or drive B and enter that drive letter. For example, if you put the Install disk in drive A enter:

A :

- If you have booted from a floppy drive, you cannot install from that drive unless you set the COMSPEC to another drive. See the information on SET in your operating system documentation.

Step 3

At the operating system A> prompt (or its equivalent), enter:

INSTALL

The installation program will display the Installation screen.

Step 4

The cursor will be at the *Source path* prompt which designates the drive from which you are installing the software. The expected source drive is your diskette drive A. You may change the source drive by typing over the information displayed.

To move to the next prompt, press [Enter] or the [↓] key.

Step 5

The *Destination Path* prompt identifies the target drive and the complete directory path where Advanced Revelation files will be installed. This must be a valid drive with free disk space greater than or equal to 5MB.

The expected target drive is your hard disk drive C and the expected directory is \AREV. You may change these by typing over the information displayed and pressing [Enter]. Advanced Revelation will copy its system files to the directory identified here and build subdirectories from it.

- ☛ If any files are present in the designated directory, they will be overwritten by the Installation process.

Step 6

When the source and target installation information is correct, press [F9] to initiate the installation process.

The installation process begins by checking the hard disk for available space. If insufficient space is available, a message will display. You can terminate the installation process by pressing [Esc] and selecting **QUIT** at the menu.

Step 7

Follow the instructions displayed by the installation program as it copies files from the diskettes. It will tell you when to put in the next diskette and identify it for you. If you inadvertently put in a diskette out of sequence, the program will prompt for the correct one.

Step 8

When the base set of Advanced Revelation files has all been copied to your hard disk, a message will display. The Installation program asks if you want to install optional modules at this time.

- ☛ Optional modules provide extensions to the basic functions of Advanced Revelation. For example, the dBASE and ASCII bonds enable you to access and update dBASE and ASCII files from within Advanced Revelation. The software for optional modules is provided on a separate disk(s) labeled to indicate what module(s) it contains.

If you want to install optional modules at this time, enter **Y**. The Install program logs you into Advanced Revelation and displays the Installation Utilities Menu. To perform the installation, follow the instructions in "Installing Optional Modules" below.

If you do not want to install optional modules at this time, enter **N**. You will be returned to the operating system. For information about logging into Advanced Revelation, turn to the “Logging In” chapter in this section.

Installing Optional Modules

Use this set of instructions when you are *first* loading Advanced Revelation onto your computer or network.

- ✎ If you are adding optional modules to an existing Advanced Revelation system (one that has been in use for any length of time) *do not follow* the instructions provided here. Instead, turn to the “Updating and Enhancing Your Advanced Revelation System” chapter and follow the instructions there.

Step 1

At the Installation Utilities Menu, press **I** to highlight the **Install** option. Then press [Enter].

Step 2

A message will display, prompting you to insert disk 1 of your module set into the diskette drive. Insert the disk and then enter the name of the source drive. For example, if you put disk 1 in drive A, enter:

A:

Step 3

Advanced Revelation displays a popup that lists the module(s) provided on this disk set. Select the module(s) that you want to install.

To choose a module, highlight it on the popup and press [Enter]. Use the [↓] and [↑] keys to move the highlight to a module you want to install and press [Enter]. Move the highlight to the next module you want to install and again press [Enter]. If you change your mind about an item, move the highlight to the item and press [Enter] again. The [Enter] key works as a toggle for choosing and rejecting items.

When you have chosen all the modules you want, press [F9] to begin the installation process.

Step 4

The Installation program displays a popup with three Upgrade Utility options. Highlight the **TEST** option and press [Enter].

The Test process reviews each disk in the current set for errors. Since the installation process cannot be stopped or reversed once it starts, this provides a way for you to check that the process will not abort due to disk errors.

Follow the instructions that appear on the screen as the test proceeds. When the test is completed, you will be returned to the Upgrade popup window.

- If you decide not to install the optional module(s), you can exit the process by pressing [Esc] at this popup. You will be returned to the Installation Utilities menu. Choose **Exit** at the menu to log out of Advanced Revelation.

Step 5

When you are ready to install the new software, choose **UPGRADE**. This choice begins the installation process. Once the process has begun, you will not be given an opportunity to stop or reverse it.

Follow the instructions that appear on the screen as the installation proceeds.

Step 6

When the information for the new software has all been copied to files on your hard disk, the Installation program displays a message and then logs you out of Advanced Revelation.

For information about logging into Advanced Revelation, turn to the "Logging In" chapter in this section.

- If you have other disk sets containing optional modules that you want to install, you can add the new modules to your new Advanced Revelation system at this time. To do so, complete the "Optional Step" below.

Optional Step

To add another module to your new Advanced Revelation system at this time, from the operating system prompt enter:

```
AREV INSTALL
```

The Installation Utilities Menu displays. Follow steps 1 through 6 above.



Chapter B 2

Updating and Enhancing Your Advanced Revelation System

This chapter explains how to use the installation utilities provided by Advanced Revelation. Use the instructions in this chapter to install add-on products that run with your existing Advanced Revelation system and to install Product Upgrades and Maintenance Updates supplied by Revelation Technologies Inc. Add-on products include Environmental Bonds, third-party software, and system utilities.

- ☛ To run the procedures in this chapter, you must have Advanced Revelation version 2.0 or above already installed on your computer or network.

About Advanced Revelation Installation Utilities . . .	B 2 . 1
Adding Optional Modules to Your System	B 2 . 1
Updating or Upgrading Advanced Revelation	B 2 . 3
Using the Installation Utility	B 2 . 4
Upgrading User Accounts	B 2 . 6
Note to the System Administrator	B 2 . 9



About Advanced Revelation Installation Utilities

You will use the options on the Advanced Revelation Installation Utilities menu to install all Advanced Revelation Upgrades and Maintenance Updates to your Advanced Revelation 2.0 system. In addition, you will use the options to install optional modules such as the dBASE and ASCII bonds and some third-party software.

The basic operation of the Installation program is the same whether you are installing an optional module or installing an Upgrade or Update. However, most product releases require some preliminary steps to prepare for the installation process. This chapter covers the following processes and topics:

- Preparing to install an optional module, including Environmental Bonds, third-party software, and system utilities. Follow the instructions in “Adding Optional Modules to Your System”.
- Preparing to install a Product Upgrade or Maintenance Update. Follow the instructions in “Updating or Upgrading Advanced Revelation”.
- Running the installation process to install new software. Follow the instructions in “Using the Installation Utility”.
- Running the Upgrade Account process to make the system changes available to existing user accounts. Follow the instructions in “Upgrading User Accounts”.
- Special information for the system administrator. See “Notes to the System Administrator”.

☞ Some third party products require additional or different procedures. Be sure to look for and follow any special instructions that are supplied with a new disk set.

There are two installation utilities:

- Use the **Install** option to install your new software release to a base set of system files in the SYSPROG account.
- Use the **Upgrade Account** option to make the changes available to user accounts.

Adding Optional Modules to Your System

Use the instructions in this section if you have previously installed a copy of Advanced Revelation 2.0 or higher. The procedure described here will add optional modules, including Environmental Bonds, third-party software, and system utilities, to your installed copy of Advanced Revelation.

☞ If you have both a Product Upgrade or Maintenance Release *and* optional module(s) that you want to install at the same time, be sure to install the Update/Upgrade first. Then install any optional modules.

☞ The Advanced Revelation Installation Utility programs can install many, but not all, third-party software products. Refer to the documentation provided with the software for information about the proper installation procedure.

When you run this process, information from the optional module disks is copied directly into your base set of Advanced Revelation system files on the SYSPROG account. Once the process is done, you will be logged out of Advanced Revelation. When you log back in, the newly installed module(s) will be available for use in the SYSPROG account. From then on, whenever you create a new account, all installed optional modules are made available to the new account automatically.

- ☛ After installing an optional module(s), you *must* run the Upgrade Account process to make the new module(s) available to users in the existing accounts on the system.

Installing an Optional Module

To install an optional module, follow these steps:

- Step 1** If you are using a network version of Advanced Revelation, make sure that only your station is using Advanced Revelation while the installation process is executing. Other stations should log out of Advanced Revelation before you begin the Installation process.
- Step 2** Log on to your existing copy of Advanced Revelation as the INSTALL user. At the operating system prompt, enter:


```
AREV INSTALL
```


The Installation Utilities menu displays.
- Step 3** Select the **Install** option at the Installation Utilities menu.
- Step 4** A message will display, prompting you to insert disk 1 of your module set into the diskette drive. Insert the disk and then enter the name of the source drive. For example, if you put disk 1 in drive A, enter:


```
A :
```
- Step 5** Advanced Revelation displays a popup that lists the module(s) provided on this disk set. Select the module(s) that you want to install and press [F9] to begin the installation process.
 - ☛ If you find that the Installation program cannot read the first disk in the new disk set, it is likely that this third-party product must be installed in another manner. Refer to the documentation supplied with your disks.

- Step 6** You are ready to use the options on the Upgrade Utility popup to complete the installation process. Follow the instructions in the section called "Using the Installation Utility", which appears later in this chapter.

Updating or Upgrading Advanced Revelation

Use the instructions in this section if you have previously installed a copy of Advanced Revelation 2.0 or higher. The procedure described here will install Product Upgrades and Maintenance Updates for your installed copy of Advanced Revelation.

When you run this process, information from the Product Upgrade or Maintenance Update disks is copied directly into your base set of Advanced Revelation system files on the SYSPROG account. Once the process is done, you will be logged out of Advanced Revelation. When you log back in, the new version of Advanced Revelation will be available for use in the SYSPROG account.

- ☛ To make the new Upgrade or Update available to users in other accounts on the system, you must run the Upgrade Account process.

Installing a Product Upgrade or Maintenance Update

To install an Advanced Revelation Upgrade or Update, follow these steps:

- Step 1** Back up your Advanced Revelation system files. The installation process will add and overwrite records in most of these files, but will leave your application files intact.

If you are unsure of how to back up only your system files, use the operating system BACKUP command or some other operating system-level backup utility to back up your entire Advanced Revelation system.

Warning ! The installation process makes permanent, irreversible changes to your copy of Advanced Revelation. If for some reason the installation process aborts or fails, your only recourse will be to restore the system from a backup copy. For information about backing up your system, refer to your operating system documentation, or see your system administrator.

You can generate a list of the files affected by the Product Upgrade or Maintenance Update before any changes are made. For details see “Using the Installation Utility”.

Step 2 If you are using a network version of Advanced Revelation, make sure that only your station is using Advanced Revelation while the installation process is executing. Other stations should log out of Advanced Revelation before you begin the Installation process.

Step 3 Log on to your existing copy of Advanced Revelation as the INSTALL user. At the operating system prompt, enter:

```
AREV INSTALL
```

The Installation Utilities menu displays.

Step 4 Select the **Install** option at the Installation Utilities menu.

Step 5 A message will display, prompting you to insert disk 1 of your disk set into the diskette drive. Insert the disk and then enter the name of the source drive. For example, if you put disk 1 in drive A, enter:

```
A:
```

Step 6 You are ready to use the options on the Upgrade Utility popup to complete the installation process. Follow the instructions in the next section, “Using the Installation Utility”.

Using the Installation Utility

Complete the steps in either “Updating or Upgrading Advanced Revelation” or “Adding Optional Modules to Your System”. Then use the instructions in this section to install your new software.

☞ If you have both a Product Upgrade or Maintenance Release *and* optional module(s) that you want to install at the same time, be sure to install the Update/Upgrade first. Then install any optional modules.

Step 1 The Installation program displays a popup with three Upgrade Utility options.

The three options at this popup are:

Option	Meaning
REPORT	This option prints a report to the screen or printer describing all the changes the upgrade will make. This report will help you decide if the installation process will interfere with customizations you have made to your Advanced Revelation system.
TEST	This option reviews each disk in the current set for errors. Since the installation process cannot be stopped or reversed once it starts, this provides a way for you to check that the process will not abort due to disk errors. The TEST option also warns if any system files needed by the upgrade are not attached. The Test process will examine the Upgrade disks one by one, displaying a report as it goes. Because it looks at every single file and record involved in the Upgrade, this process takes several minutes.
UPGRADE	This choice begins the installation process. Once the process has begun, you will not be given an opportunity to stop or reverse it.

- ☞ If you decide not to install the Upgrade, Update, or optional module, you can exit the process by pressing [Esc] at this popup. You will be returned to the Installation Utilities menu.

Step 2 (optional) Choose **REPORT** to print a report that itemizes all of the changes the upgrade will make. When the report is completed, you are returned to the Upgrade popup.

- ☞ If you decide that the new software will overwrite changes you have made to the system files, you must save your changes to a safe location. Install the upgrade, and then reinstall the changes you require.

Step 3 Choose **TEST** to verify that the current disk set is usable.

Follow the instructions that appear on the screen as the test proceeds. The process will tell you when to put in the next disk and identify it for you. If you inadvertently put in the wrong disk, the program will alert you. When the test is completed, you are returned to the Installation Utilities menu.

Warning! The Test process warns if a system file is not attached. In this case, you *must* attach the file before proceeding with the upgrade.

Do not attempt to do a partial upgrade, or to abort the Upgrade process before it is through. If you do so, your system will not be fully upgraded and may not work properly.

Step 4 When you are ready to install the new software, choose **UPGRADE** to start the procedure.

Follow the instructions that appear on the screen as the upgrade proceeds. The program will tell you when to put in the next disk and identify it for you.

Step 5 When the information for the new software has all been copied to files on your hard disk, the Installation program displays a message and then logs you out of Advanced Revelation.

Step 6 If you have other disk sets containing optional modules that you want to install, you can add the new modules to your base set of system files in the SYSPROG account at this time. See the instructions in “Adding Optional Modules to Your System” earlier in this chapter.

Step 7 When you are finished installing the Advanced Revelation Upgrade or Update, and/or any optional module(s), you must run the Upgrade Account process to make the system changes available to existing user accounts. Follow the directions in the next section “Upgrading User Accounts”.

Upgrading User Accounts

Advanced Revelation’s Installation utility makes changes to a base set of system files in the SYSPROG account. The “base set” of files consists of the Advanced Revelation system files that are attached at the time that the new software is installed. Following the installation of an Advanced Revelation Product Upgrade or Maintenance Update, and/or optional modules, you should run the Upgrade Account process to update files in existing user accounts. There are two ways to go about this:

- Log on to SYSPROG and update files in all (or selected) user accounts.
- Log on to a user account and update the currently attached files.

Upgrading User Accounts from SYSPROG

Following any modification to the system, you must update existing user accounts to make the changes available. The Advanced Revelation Upgrade Account utility enables you to add the latest changes to your files in user accounts. When you log on to the SYSPROG account, the Upgrade Account utility offers three ways to update these files:

ALL ACCOUNTS Use this option to update all non-SYSPROG accounts in one pass. The process searches for all current accounts, allows you to select the accounts that you want to upgrade, and then makes all necessary changes.

A list of the files to be updated will display. For each account, ALL ACCOUNTS will prompt you to enter the location or volume name where the current version of each file resides (the default is REVBOOT). You can thus update the appropriate files for each account regardless of where the files are kept.

ATTACHED FILES This option enables a user logged in to SYSPROG to upgrade the versions of system files currently attached, making all necessary changes automatically.

If you choose this option, the Upgrade Account program will use the base set of files on the SYSPROG account as the basis for updating your local version of these files.

To use this option, you will log in as a specific user in the SYSPROG account. Then run the **ATTACHED FILES** option to update the set of system files that are currently attached.

A list of the files to be updated will display. Enter Y at the prompt to confirm that you want to update the attached files. If the Upgrade Account process determines that a file has been updated already, it will not be updated again.

SINGLE FILE This option permits you to update a local version of any system file that the Installation process normally updates. For example, if you have a copy of the MESSAGES file that was not accessible to the Installation process, you can update it using this option.

The SINGLE FILE option does not update the dictionary of a file automatically. To update the dictionary of a file, you will need to run the SINGLE FILE process twice, once for the file and once for its dictionary.

The **SINGLE FILE** option updates the version of a file that you currently have attached. The Upgrade Account program will use the base set of files in SYSPROG as the basis for updating your local version of this file.

- ☛ If the Upgrade Account process determines that a file has been updated already, it will not be updated again.

Executing the Upgrade Account Process in SYSPROG

To execute the Upgrade Account process, follow these steps:

- Step 1* Make sure that no users are logged in to accounts that you intend to update. Other users should log out while the upgrade is executing.
- Step 2* Log into Advanced Revelation on the SYSPROG account. Select the **Management-Accounts-Upgrade Accounts** options from the Advanced Revelation Main menu.
 - ☛ **Upgrade Account** is also an option on the Installation Utilities Menu. Enter AREV INSTALL at the operating system prompt to access this menu.
- Step 3* A popup will display offering you the three options described above: **ALL ACCOUNTS**, **ATTACHED FILES**, and **SINGLE FILE**. Choose the option you wish to execute.

If you decide not to perform an account upgrade, you can exit the process by pressing [Esc] at this popup. You will be returned to the menu.
- Step 4* After determining that the right user accounts have been updated, you can delete the UPDATE.LOG and UPDATE.CTL files that are in the subdirectory containing the AREV.EXE file. For more information, see the next section, "Note to the System Administrator".

Executing the Upgrade Account Process in a non-SYSPROG Account

Following the installation of new software, this procedure enables an individual user to log on to Advanced Revelation and upgrade the set of files that they normally use. To execute the Upgrade Account process, follow these steps:

- Step 1* Log on to the account as you normally do.
- Step 2* Select the **Management-Accounts-Upgrade Accounts** options from the Advanced Revelation Main menu.

Step 3 A message will display, listing all of the files that will be upgraded. Enter Y at the prompt to begin the upgrade process. When this automatic process is complete, you are returned to the Account Management menu.

If you decide not to perform an account upgrade, you can exit the process by entering “N” at the prompt. You will be returned to the Account Management menu.

- If there is a password on the SYSPROG account, you cannot run the Upgrade Account process from a non-SYSPROG account. A message will report that the process could not attach to a file in the SYSPROG account. When the message clears, you are returned to the Account Management menu.

Note to the System Administrator

In order to use the Advanced Revelation Installation utilities most effectively, you should be aware of how the Installation Utilities function and how they alter or affect your system. The topics below address important information that you should review.

About the UPGRADE Files

The Install program maintains two operating system files — UPGRADE.CTL and UPGRADE.LOG — in the subdirectory that contains your AREV.EXE file.

- The UPGRADE.CTL file contains the location of each file in the base set of files in the SYSPROG account. The “base set” of files consists of the Advanced Revelation system files that were attached at the time that the most recent Product Upgrade, Maintenance Update, and/or optional module was installed.
- The UPGRADE.LOG file contains information about all of the records and files that need to be copied from the base set of system files in SYSPROG to the system files in individual user accounts.

The Upgrade Account program uses the information in UPGRADE.CTL and UPGRADE.LOG to make updates to the appropriate files in user accounts.

The way that the Install program maintains the UPGRADE.CTL file and UPGRADE.LOG file depends on the software that you are installing:

- New UPGRADE.CTL and UPGRADE.LOG files are created each time you install a Maintenance Update or Product Upgrade supplied by Revelation Technologies, Inc. If an UPGRADE.CTL file and UPGRADE.LOG file already exist in the subdirectory containing the AREV.EXE file, they are overwritten by the Installation program.

- When you install an optional module, such as an Environmental Bond or a third-party application, the installation process adds information to the existing UPGRADE.CTL and UPGRADE.LOG files. Installing optional modules does not overwrite these files if they exist. If UPGRADE.CTL and UPGRADE.LOG files do not exist, the installation process creates them.

Once UPGRADE.CTL and UPGRADE.LOG are created by the Installation program, the information needed to perform user account upgrades is available until another Maintenance Update or Product Upgrade is installed. At that time, the current UPGRADE.CTL and UPGRADE.LOG are overwritten with new information.

Warning ! If considerable time lapses between the installation of an Upgrade/Update and the installation of an optional module, you are advised to delete the UPGRADE.CTL and UPGRADE.LOG before installing the optional module. If you do not, when the Upgrade Account process runs, it will copy all of the records that were added and changed by the last Upgrade/Update as well as the records and files added by the new optional module.

If an account has already been upgraded with the latest Upgrade/Update records, this is a waste of time. And, in cases where the UPGRADE.LOG directs the program to create a new file when one already exists, there could be loss of data.

Updating Existing User Accounts

Following any modification to the system, you must update existing user accounts to make the changes available. There are two ways to go about this:

- The System Administrator can log into SYSPROG and update all (or selected) user accounts.
- The Account User can log on as usual and update the current account.

The safest way to provide the latest installed versions of your Advanced Revelation software and optional modules to your users is to run the Account Update program from the SYSPROG account. Once you have upgraded all accounts, delete the UPGRADE.CTL file and UPGRADE.LOG file in REVBOOT.

Alternately, you can notify your users that there is a new optional module or upgrade available. Interested users can log in to accounts as they normally do and select the **Management-Accounts-Upgrade Account** option from the Advanced Revelation Main menu. This calls a program that upgrades files in the current account to make the latest changes available. After all users have upgraded their accounts, you should delete the UPGRADE.CTL file and UPGRADE.LOG file in REVBOOT.

- ✎ If you want users to be able to update non-SYSPROG accounts using the **Management-Accounts-Upgrade Account** option from the Advanced Revelation Main menu, you must remove any SYSPROG password.

If you have a password on the SYSPROG account, the Upgrade Account process will be unable to read from the base files in the SYSPROG account, and will not perform the Upgrade.

Installing on a Networking

In order to install software on a network, you must have the ability to create new files and subdirectories and to read and write to them.



Chapter B 3

Removing Optional Modules

This chapter explains how to remove optional modules that were previously added to you Advanced Revelation system using the Advanced Revelation Installation utility.

Removing Optional Modules B 3 . 1



Removing Optional Modules

Use the procedure described in this chapter if you want to remove an installed Environmental Bond, third-party software, or an add-on system utility

When you run the Remove process, it will remove the module and the routines that support it from your base set of Advanced Revelation system files on the SYSPROG account. The “base set” of files consists of the Advanced Revelation system files that are currently attached, including all global files such as COMMANDS, SYS.POPUPS, SYS.MENUS, etc.

Once the process is done, you will be logged out of Advanced Revelation. When you log back in, the removed module(s) will no longer be available for use in any account in Advanced Revelation.

- ☛ The Remove process only removes optional modules that were installed using the Advanced Revelation Installation Utility programs. The Installation process is described in the “Updating and Enhancing Your Advanced Revelation System” chapter.

The Optional Module Removal Procedure

To remove an optional module, follow these steps:

- Step 1** If you are using a network version of Advanced Revelation, make sure that only your station is using Advanced Revelation while the Remove process is executing. Other stations should log out of Advanced Revelation before you begin the Remove process.
- Step 2** Log on to your existing copy of Advanced Revelation as the INSTALL user. At the operating system prompt, enter:

AREV INSTALL

The Installation Utilities menu displays.
- Step 3** Select the **Remove** option at the Installation Utilities menu.
- Step 4** Advanced Revelation displays a popup that lists the optional module(s) that are currently available on your Advanced Revelation system. Select the module(s) that you want to remove and press [F9] to begin the removal process.
- Step 5** Advanced Revelation will identify the module that it is currently removing. Another message displays when the removal is complete. You are automatically logged out of Advanced Revelation.



Chapter B 4

Logging On to Advanced Revelation

This chapter explains how to enter Advanced Revelation by logging on. It also describes the use of logon options to invoke special Advanced Revelation processing modes.

Logging On to Advanced Revelation B 4 . 1



Logging On to Advanced Revelation

Once Advanced Revelation has been installed, you may log on. To do so, access the subdirectory containing the Advanced Revelation files and the AREV.EXE file, and invoke Advanced Revelation.

For example, if you installed Advanced Revelation using the defaults in the Installation process, make sure you are in the C:\AREV directory, and then invoke Advanced Revelation. From the operating system prompt enter:

```
AREV
```

Using Logon Options

Depending on your computer's hardware and software configuration, you may want to log on to Advanced Revelation using one or more logon options. The logon options invoke special Advanced Revelation modes that support a variety of processes. For example, if your system has expanded memory capabilities conforming to LIM EMS 4.0, there are two logon options that you can use to support the expanded memory device driver.

The E Option In certain isolated circumstances, Advanced Revelation does not receive a correct response when checking for a math coprocessor. If this is the case, these computers will sometimes "hang" or fail to continue processing and may display the error message "Divide overflow" or "Stack overflow". To correct the problem, append /E to your logon command. This causes Advanced Revelation to load a software emulator of the math coprocessor.

For example, to log on after completing the installation, type:

```
AREV /E
```

The S Option To suppress the logon banner that normally displays while the system is loading, append an S to your logon command. For example,

```
AREV /S
```

Additional Logon Options Logon options for LIM EMS 4.0 are explained in the "Using Expanded Memory Support" chapter in this section.

If you are using an Environmental Bond, check the "Description" topic in your bond documentation to determine if there are logon options used to support the bond.

Command Syntax for Logon Options To specify one or more options in the logon command, type in the command as usual. Then add a / (forward slash) at the end of the command to indicate that options will follow.

Following the / character, type in the desired option(s). Separate each option with a space. For example,

```
AREV SYSPROG /E H50 X
```

The command shown above logs the user into the SYSPROG account, turns on math coprocessor emulation, reserves 50 file handles for use in Advanced Revelation and specifies that expanded memory is used in Expanded mode.

Setting the Help Level

The first time you log on to Advanced Revelation, you will see a popup window that asks you to set a help level. The popup also offers you an opportunity to read an overview of Advanced Revelation. The popup window looks like this:

Help Level Selection		
	Help	Description
1	>	1 Novice
2		2 Intermediate
3		0 Advanced
4		I An Introduction to Advanced Revelation

Along with the popup you will see a message welcoming you to Advanced Revelation. The message contains information about using Advanced Revelation, and explains the choices in the popup window.

Use the arrow keys or the [PgUp] and [PgDn] keys to see other pages of the message. When you have finished reading the message, press [Esc]. At that point you can make a choice from the popup window.

To make a selection from the popup, use the arrow keys to move the highlight bar. Set the highlight bar on your choice, and press [Enter].

Setting the help level to 1 (novice) causes Advanced Revelation to present help messages automatically, and list active function keys in the status line. A help level of 2 (intermediate) provides a status line listing active function keys, but displays help messages only on demand. A help level of 0 (advanced) provides help messages only on demand, and displays the full status line.

For more information on the use of help levels, see the “Using System Features” chapter in *Getting Started*.

- ☛ Advanced Revelation will detect whether your computer has a monochrome or color/graphics monitor. If you do not specify a particular user or account name when logging in, Advanced Revelation will choose one for you.

If you have a color or graphics board, Advanced Revelation will log you in as the user COLOR.USER. If you have a monochrome monitor, you will be logged in as the user USER.

Chapter B 5

Using Expanded Memory Support

Advanced Revelation supports the use of expanded memory software conforming to LIM EMS 4.0. This chapter provides information on how to configure your system and how to invoke expanded memory support.

Introduction	B 5 . 1
System Requirements	B 5 . 1
Using Expanded Memory Logon Options	B 5 . 2
Managing Expanded Memory	B 5 . 4
Displaying the Status of Expanded Memory	B 5 . 5
System Configuration	B 5 . 6



Introduction

If your system is configured correctly for expanded memory, Advanced Revelation can take advantage of this resource.

Benefits of Expanded Memory

Advanced Revelation's expanded memory support provides additional workspace in RAM for applications. Programmers can take advantage of expanded memory to maintain a greater number of large character strings and a greater number of large subroutines. This capability is especially important in sophisticated applications that use many supporting routines or large records, or in networked applications that have less memory available.

- Expanded memory support does *not* allow you to create character strings larger than 64K. Expanded memory also does *not* increase the maximum number of variables, nor does it increase the maximum number of programs. Programmers will still need to exercise caution when using dimensioned arrays and labelled common areas.

Advanced Revelation's Use of Memory

Advanced Revelation will use all available conventional memory (conventional memory is the area of RAM up to 640K). If you are using expanded memory manager (EMM) software, Advanced Revelation can use expanded memory (the memory above 1 megabyte).

When expanded memory support is invoked, Advanced Revelation will automatically allocate half of the system's available expanded memory for its use, up to a maximum of 4 megabytes. You can use a special logon option to increase or decrease the amount of expanded memory that is initially allocated. For more information, see "Using Expanded Memory Logon Options" below.

System Requirements

To take advantage of expanded memory, you will need:

- A minimum of 256K expanded memory, either through hardware or through software emulation.
- Expanded Memory Manager (EMM) software that conforms to the Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS) version 4.0. The EMM software must support all the advanced functions defined in the LIM EMS 4.0.
- At least 256K free expanded memory available when you start Advanced Revelation.

Implementation of expanded memory support is optimized for use with 386-based machines.

Using Expanded Memory Logon Options

Logon options enable you to control how your expanded memory support is used in Advanced Revelation. Available logon options and logon command syntax are explained below.

Invoking Expanded Memory Capability Modes

Advanced Revelation can use expanded memory in three different modes. The mode is determined by the way in which you log on to Advanced Revelation.

Programs-only Mode If your system is configured correctly for expanded memory, Advanced Revelation uses it automatically when you log in. By default, expanded memory is invoked in Programs-only mode.

In the Programs-only mode, programs that are longer than 508 bytes of object code will be swapped out to expanded memory as necessary.

This does not mean that you can load more programs, only that you can load larger programs.

Expanded Mode Expanded mode is invoked by using the X option in your logon command at the operating system prompt. For example:

```
AREV SYSPROG /X
```

In Expanded mode, *any* string of characters longer than 508 bytes, including programs, will be swapped out to expanded memory as necessary. Expanded mode operates faster than Programs-only mode and decreases the chance of running out of conventional memory.

Expanded mode does have this limitation: C and assembly language routines can have, at most, two valid pointers into Advanced Revelation string space. The two most recently fetched pointers are guaranteed to be valid. Using more than two pointers will yield unpredictable results.

All C and assembly language routines that are provided as part of your Advanced Revelation system comply with this requirement. To run using Expanded mode, any application that you write or run under Advanced Revelation must abide by this limit.

Warning ! Do not use Expanded mode unless you are sure that the C and assembly language routines executed by your Advanced Revelation application have no more than two pointers into Advanced Revelation string space at any given time. Your developer can tell you whether your application complies with Expanded mode limitations.

Off Mode In some situations you may want to log on to Advanced Revelation and not invoke expanded memory support. Use the O option in your logon command at the operating system prompt to turn off use of Advanced Revelation's expanded memory support. For example:

```
AREV SYSPROG /O
```

Controlling Initial Expanded Memory Allocation

By default, when expanded memory capability is invoked, Advanced Revelation allocates half of the system's available expanded memory for its use, up to a maximum of 4 megabytes. You can use a special logon option to increase or decrease the amount of expanded memory that is initially allocated.

To allocate a specific amount of expanded memory for Advanced Revelation, use the M option in your logon command at the operating system prompt. For example:

```
AREV SYSPROG /Mnnnn
```

where:

nnnn is the number of kilobytes of expanded memory that Advanced Revelation initially allocates for use.

Advanced Revelation requires at least 256K of free expanded memory at logon. If your M option specifies less than 256K, Advanced Revelation will allocate at least 256K of expanded memory.

The maximum number of kilobytes that Advanced Revelation will ever allocate is 4096 (4 megabytes). If you enter a number that is greater than the available expanded memory, Advanced Revelation allocates all of the memory that is currently available.

- At logon, if you display the WHO window, you may find that the *EM Allocated* prompt displays a slightly lower number than the one that you specified in your M logon option. This is because expanded memory is allocated in 64K increments.

Command Syntax for Logon Options

To specify one or more options in the logon command, type in the command as usual. Then add a / (forward slash) at the end of the command to indicate that options will follow. Following the / character, type in the desired option(s), separating each option with a space. For example:

```
AREV SYSPROG /X M512
```

The command shown above logs the user into the SYSPROG account, specifies that expanded memory is used in Expanded mode, and initially allocates 512K of expanded memory for use by Advanced Revelation.

Managing Expanded Memory

To support expanded memory, Advanced Revelation requires three blocks of memory:

- **One 16K expanded memory window.** Because systems such as Advanced Revelation cannot access expanded memory directly, they require the use of “mappable” regions which they can access. Mappable regions are controlled by the expanded memory manager and are made available to applications for use as windows into expanded memory.
- **Two 64K overflow buffers.** The overflow buffers are used to prevent data in programs from shifting in memory during critical operations. These buffers may be allocated in mappable regions above 640K if enough mappable memory is available.

☛ To determine where Advanced Revelation has allocated the 16K expanded memory window and 64K overflow buffers, use the WHO window. For more information, see “Displaying the Status of Expanded Memory” later in this chapter.

Performance will be improved significantly when one, two, or all three of these blocks are allocated in mappable regions, also known as swapping regions, residing above 640K and below 1M (non-conventional memory).

If mappable regions are not found for the 64K overflow buffers, the buffers will be allocated in conventional memory, and the 16K expanded memory window will be allocated in the *page frame*. The page frame is a 64K mappable region provided by the expanded memory manager and used to access expanded memory. This not only wastes 48K of the page frame, it also has the undesirable side effect of reducing conventional memory by as much as 128K and possibly reducing the number of variables that can be used.

Therefore, if your expanded memory manager is capable of supporting additional mappable regions, you can improve performance significantly by making as much mappable memory available as possible.

☛ If your expanded memory manager supports only one 64K mappable region, namely the 64K page frame, it will be impossible to increase the number of mappable regions.

To make more mappable memory available for Advanced Revelation to use, avoid filling non-conventional memory above 768K. Some expanded memory managers provide the capability of filling unused portions of non-conventional memory with usable high RAM (also called high DOS memory). However, Advanced Revelation cannot directly take advantage of high RAM, so you may want to convert some or all of the high RAM to mappable memory. See your EMM software documentation for details.

- ✎ Some expanded memory managers may not recognize that areas of non-conventional memory are being used by other peripherals (network cards, for example). When there is a conflict between your EMM and a peripheral device over available non-conventional memory, you may be unable to log in to Advanced Revelation. Your EMM software documentation should provide information about how to detect and resolve this problem.

Displaying the Status of Expanded Memory

For those using expanded memory, the WHO window provides information about Advanced Revelation's use of memory. Enter:

WHO

at TCL to display the WHO window. The WHO window is for information only — you cannot make changes at this window.

On the first page of the WHO window, note the information at the *Available memory* and *Expanded Memory* prompts. From the first page of the WHO window, press [Pg Up] to display page three of the WHO window, which is labeled "EXPANDED MEMORY INFORMATION". When EMM software is active, information displays at the *EM Used*, *EM Allocated*, *16K EM Window*, *64K Overflow Buffer 1* and *64K Overflow Buffer 2* prompts:

Available memory

The number of bytes currently available to Advanced Revelation in conventional memory displays.

Expanded Memory

If Advanced Revelation recognizes and is using EMM software, the word "Active" displays. If no EMM software is recognized, the word "Inactive" displays.

EM Used

The number of bytes in expanded memory that Advanced Revelation is currently using displays.

EM Allocated

The number of bytes in expanded memory that is currently allocated to Advanced Revelation displays.

- ☛ By default, Advanced Revelation initially allocates to itself only half of the available expanded memory. If you use the M logon option, Advanced Revelation allocates the amount of expanded memory that you specified at logon. In either case, when more expanded memory is needed, Advanced Revelation will secure additional memory in 64K byte segments.

16K EM Window

In order to support expanded memory capabilities, Advanced Revelation must have a 16K mappable segment available. The starting segment address of the 16K mapped segment displays at this prompt.

64K Overflow Buffers

To support expanded memory, Advanced Revelation makes use of two 64K overflow buffers. The starting segment addresses for the two 64K overflow buffers display at these prompts.

System Configuration

Sample Configuration for a 386-Based System

The EMS software listed below conforms to the requirements for Advanced Revelation:

- 386^{MAX}
software-emulated expanded memory from Qualitas, Inc.
- Quarterdeck Expanded Memory Manager
software-emulated expanded memory from Quarterdeck Office Systems

This section describes how to create an efficient memory configuration for Advanced Revelation using the Quarterdeck Expanded Memory Manager 386 (QEMM).

- ☛ These instructions do not attempt to replace your expanded memory software or hardware documentation. In fact, it is assumed that you have access to your system's documentation and that you are familiar with its terms and configuration procedures.

To follow this suggested setup, you must have:

- 1 megabyte of extended memory
- Quarterdeck Expanded Memory Manager 386 (QEMM) installed on a 386-based computer

Installing QEMM

Install QEMM, following the documented procedure supplied with your memory manager software. When using QEMM command line options, do not use the RAM option.

Checking for EMM Recognition

Verify that Advanced Revelation can recognize and use the expanded memory manager. From the operating system prompt, log on to Advanced Revelation as usual. For example:

```
AREV SYSPROG
```

Press [F5] to display the TCL window. At the TCL window, enter:

```
WHO
```

to display the WHO window. If Advanced Revelation recognizes and is using the expanded memory manager, the word "Active" will display at the *Expanded Memory* prompt.

Troubleshooting

If you could not log in to Advanced Revelation, QEMM may not recognize that areas of non-conventional memory are being used by other peripherals (network cards, for example). Refer to your QEMM documentation for information about how to eliminate conflicts over available non-conventional memory between QEMM and peripheral devices.

If the *Expanded Memory* prompt at the WHO window does not display "Active", check the following:

- Reboot your system and watch the screen for any error messages indicating that the expanded memory manager is not installed correctly. For example, you may see a message stating that there are errors in the CONFIG.SYS file.
- Verify that you are running a version of QEMM that supports the LIM EMS 4.0 specification
- QEMM must provide at least 256K of available expanded memory. An error in your setup may reduce the amount of available expanded memory. When this is the case, Advanced Revelation cannot recognize or use the expanded memory capabilities.

Examining Your Current Configuration

After ensuring that Advanced Revelation is using expanded memory, check the segment addresses (shown in hex) displayed in the WHO window next to the prompts *16K EM Window*, *64K Overflow Buffer 1*, and *64K Overflow Buffer 2*. For maximum efficiency, all three addresses should be greater than or equal to C000.

Fine Tuning Your Configuration

If you have an EGA adaptor, you may be able to improve your existing configuration.

Since Advanced Revelation does not use the EGA modes, memory from A000-AFFF can be used for conventional memory, providing Advanced Revelation with an additional 64K of memory.

1. Use QEMM's option `INCLUDE=A000-AFFF` on the QEMM command line in your `CONFIG.SYS` file.
2. Reboot your computer.
3. Execute the program `NOEGA.COM` which is supplied by Quarterdeck.
4. Type QEMM at the operating system prompt, and use the screen display to confirm that the address range 0000-AFFF is assigned to conventional memory. You should have 704K of conventional memory.

Once you are familiar with Advanced Revelation's use of non-conventional memory, you may be able to make even more memory available by experimenting with the RAM option. Use the RAM option to convert into High RAM any regions in non-conventional memory that Advanced Revelation is not using. Then use the `LOADHI.COM` program supplied by Quarterdeck to move TSR (terminate and stay resident) programs and device drivers out of conventional memory and into the regions in non-conventional memory that you have defined as High RAM.

Suggested Configuration for a 286-Based System

Advanced Revelation will perform best on a 286-based computer that uses EMS software that supports at least 5 pages for mapping. Software emulators of expanded memory for 286-based machines are generally limited to one 64K page frame (which is only 4 mappable pages). Hardware expanded memory usually provides a minimum of 5 mappable pages, so hardware expanded memory is likely to produce a more efficient system when used with Advanced Revelation.

The expanded memory managers listed below conforms to the requirements of Advanced Revelation:

- XMA2EMS running DOS 4.0
software-emulated expanded memory from IBM
- ABOVE BOARD
hardware expanded memory from Intel
- Rampage
hardware expanded memory from AST Research, Inc.

Chapter C 1

Introduction to the Online Tutorial

Advanced Revelation's online Tutorial is the perfect way to learn how to use Advanced Revelation. Once you get started, almost everything you need to know is right there on the screen for you. As a result, this chapter simply tells you how to start and end the tutorial and explains the keystrokes that you will use throughout.

Introduction to the Online Tutorial	C 1 . 1
Accessing the Tutorial	C 1 . 1
Running a Tutorial Lesson	C 1 . 5



Introduction to the Online Tutorial

If you are a new user of Advanced Revelation, the perfect place to begin using Advanced Revelation is the online Tutorial. The Tutorial provides you with a complete overview and guided tour of the major capabilities of Advanced Revelation. Even if you have used Advanced Revelation before, you can take advantage of the Tutorial. If you can't remember how to accomplish a task, you can have the Tutorial *show* you how.

The Tutorial consists of about two dozen self-contained lessons. Each lesson shows you one major feature of Advanced Revelation. For example, one lesson shows you how to paint a new window, while another explains how you can create a new menu. As each lesson proceeds, the Tutorial makes sure that you enter the proper keystrokes for the task you are accomplishing — you cannot make a mistake.

Each lesson takes about five to ten minutes to run. If you like, you can speed up or slow down the Tutorial to suit your own taste.

The Tutorial is written entirely in Advanced Revelation. As you run the Tutorial you will be executing the same commands and entering the same keystrokes that you would if you were building and using an application. In fact, you can even interrupt the Tutorial at any point and explore Advanced Revelation on your own — the files, windows, records, and data that you create in the Tutorial will all be available.

Because each lesson is self-contained, you can start anywhere you like. If you are interested in one of the later lessons, begin there. The Tutorial makes sure that everything you need is available for each lesson. You can repeat each lesson as often as you like.

The instructions below explain how to start and end the Tutorial, and explain what keystrokes you need for the Tutorial. Most of what you need for the Tutorial, however, will be right on the screen in front of you. The best way to learn about the Tutorial — and about Advanced Revelation — is to start it right up.

Accessing the Tutorial

You can access the Tutorial any time that you are logged in to Advanced Revelation. This section explains how to enter and exit the Tutorial and how to start and stop a Tutorial lesson.

Logging on to Advanced Revelation

To log on to Advanced Revelation, begin at the operating system prompt. If you are not already there, change to the drive or subdirectory where Advanced Revelation is installed. Then enter AREV to log on to Advanced Revelation.

For example, if you have installed Advanced Revelation on the C:\AREV subdirectory, you might enter these commands from the operating system prompt:

```
C:
CD \AREV
AREV
```

- ☛ If you normally log on to Advanced Revelation using an account or user name, continue to do so. You can start the Tutorial as described below.

Starting the Tutorial

Once you log on to Advanced Revelation, there are two ways to access the Tutorial

- From the Advanced Revelation Main menu.
 - From The Command Level window (TCL).
- ☛ Only one user can run the Tutorial at a time. If you try to access the Tutorial while another person is running it, a message similar to the one below displays. You are returned to the Advanced Revelation Main menu or to TCL.

```
The Tutorial is in use elsewhere.
      Try again later.
...Press any key to continue...
```

Starting the Tutorial from a Menu

You can display the Advanced Revelation Main menu by pressing [F10]. At the Advanced Revelation Main menu, select the options **Access-Tutorial**.

The Tutorial begins and the first Tutorial message displays. Press any key to clear this message from your screen and display the Tutorial Lessons popup.

Using a Menu

All menus in Advanced Revelation work alike. To choose an option, highlight it and press [Enter] or [F9] (Save). Use any of these methods to move the highlight:

- To go directly to any option, type the first letter of the option you want.
- To go to the next option in a list, press [Spacebar].
- To move through the options, use the arrow key that points in the direction you want to move.
- To go to the first option, press [Home].
- To go to the last option, press [End].

A brief message displayed at the bottom of the menu explains the purpose of the highlighted option.

Access from the TCL Window You can display the Command Level window (TCL) by pressing [F5]. At the TCL window, enter:

TUTORIAL

The Tutorial begins and the first Tutorial message displays. Press any key to clear this message from your screen and display the Tutorial Lessons popup.

Starting a Lesson

The Tutorial Lessons popup lists all of the lessons provided in the online Tutorial. This is a two-page popup. Only part of the lessons are listed on page one. Press [PgDn] to see the lessons listed on the second page. Press [PgDn] again to redisplay the first page.

To start one of the lessons listed on the popup, type the number of the lesson and press [Enter].

Use this popup to start any lesson. Each lesson stands alone — it is not necessary to complete them in any particular order.

Stopping a Lesson

You can interrupt any lesson by pressing the [Esc] key. When you press [Esc] a message like this one will display:

```
Leave the Tutorial lesson?
Are you sure?
(Y/N)
```

Enter “Y” to exit to the Tutorial Lessons popup. If you want to continue the current lesson, enter “N”. The lesson will resume.

Exiting the Tutorial

While running the Tutorial, you can quit at any time. If you prefer, you can temporarily interrupt a lesson to pursue another task in Advanced Revelation and then return to resume the lesson where you left off.

Leaving the Tutorial Temporarily

Advanced Revelation is designed for multilevel activity: you can interrupt one task, pick up another, then return to your place with a few quick keystrokes.

Both the [F10] (Menu) and [F5] (TCL) keys enable you to stop a Tutorial lesson temporarily to try out a process in Advanced Revelation, and then return to resume a lesson. This means that you can use the TCL window or the Advanced Revelation

menu system for some hands-on practice with the tools you are learning about.

To suspend a Tutorial lesson and pursue another task, press [F5] to go to The Command Level (TCL) or press [F10] (Menu) to enter the menu system. The Tutorial displays a message similar to this one:

```
Temporarily leave the Tutorial?  
Are you sure?  
(Y/N)
```

Type Y and press [Enter] to confirm that you want to leave. If you want to continue the current lesson without interruption, type N and press [Enter]. The lesson will resume.

- ✚ If your environment does not allow you to access TCL, you will not be able to use [F5] to display the TCL window.

When you are ready to return to the suspended lesson, press [Esc] to return to the Tutorial level.

When you leave the Tutorial temporarily, Advanced Revelation continues to view you as a Tutorial user. This means that Tutorial files are substituted for some of the files that are normally available to you. For example, when you use Paint, you will access the Tutorial's TEMPLATES file.

Warning ! To preserve the integrity of the Tutorial, do not make any changes to existing records in Tutorial files. Doing so may cause unpredictable results when the Tutorial is run.

Quitting the Tutorial

You can quit the Tutorial at any time by pressing [Esc]. Stop the current lesson as described above. Then press [Esc] at the Tutorial Lessons popup to leave the Tutorial.

Depending on how you started the Tutorial, you will return to Advanced Revelation at either the Advanced Revelation Main menu or at the TCL window. When you quit the Tutorial in this way, the Tutorial Lessons popup is no longer available unless you restart the Tutorial.

- ✚ When you enter the Tutorial, Tutorial files are substituted for some of the files that are normally available to you. To regain access to the set of files that you

usually use, you must log out of Advanced Revelation and log back on to the system.

Logging Out of Advanced Revelation

You can leave Advanced Revelation either through the Exit menu or by entering the OFF command directly at The Command Level (TCL).

- To leave through the Exit menu, choose the options **Exit-Logoff** from the Advanced Revelation Main menu.
- To enter a command at TCL, press [F5] (TCL) from anywhere in Advanced Revelation and enter the command on the command line.

Running a Tutorial Lesson

The lessons in the online Tutorial use a series of messages and screens to show you how Advanced Revelation works. You can press special function keys to get more help and to control the speed at which the lessons run.

Getting Online Help

There are two help keys that you can press in order to see more information about the current process.

Detail (Prompt) Help In some lessons, the Tutorial asks you to make an entry at a prompt or make a selection from a popup. If you want more information, you can press [F1] to display detailed help text. This help text is provided by the Advanced Revelation online help system.

Concept Help To see information about the process a window or tool performs, press [Ctrl-F2]. Throughout Advanced Revelation you can rely on [Ctrl-F2] for information that will help you to run a process and tell you what to expect when you do.

Clearing Tutorial Messages

Much of the information in the Tutorial is presented in messages that display on your screen. Each of the messages is set to display for a fixed amount of time. When the time elapses, the current message clears automatically and the next message or action in the lesson occurs.

You can increase the pace of the Tutorial by clearing any message yourself. Press any key to clear the current message and move on to the next step in the lesson.

Entering Keystroke Responses

As a lesson progresses, you will be asked to enter specific keystrokes. Directions about what keys to press will display near the bottom of the screen in a message response window that looks similar to this one:

```
Press the following keys:  
CUSTOMER[Enter]
```

When a message like this displays, the Tutorial lesson waits for your keystrokes. Each keystroke is checked as you enter it. If you enter a keystroke that is unacceptable, a beep will sound to indicate that the wrong key was pressed.

- ☛ The keystroke checker is case sensitive, so each keystroke must be entered *exactly* as specified.

Controlling Keystroke Playback Speed

Sometimes a Tutorial lesson will automatically play a series of keystrokes for you. The [+] and [-] keys enable you to control how fast the keystrokes play back.

The speed control keys are available any time a Tutorial message displays. To increase the speed of keystroke playback, press the [+] key. To decrease the speed of keystroke playback, press the [-] key.

Each time you press the [+] or [-] key it speeds up or slows down the playback speed by 1/100 of a second. By holding the [+] or [-] key down, you can make a significant change.

Adjustments that you make to playback speed will remain in effect during the current session in Advanced Revelation. When you log out of Advanced Revelation the playback speed returns to the default setting.

Summary of Active Keys

The table that follows lists the keys that you can use when running the Tutorial.

When you want to:	Press this key:
Return to the Tutorial Lessons popup or quit the Tutorial	[Esc]
Go to The Command Level (TCL)	[F5]
Go to the Main menu	[F10]
Read help text about the current prompt or popup	[F1]
Read help text about the concept behind the current window or process	[Ctrl-F2]
Speed up demonstration keystroke playback	[+]
Slow down demonstration keystroke playback	[-]



Chapter D1

Getting Started

This chapter covers the basics of the Advanced Revelation environment so you can start using the system. The “Using System Features” chapter that follows provides more detailed information about Advanced Revelation’s menu system and status lines and explains how to use windows as you create your application.

Starting, Moving Around, and Leaving	D 1 . 1
Getting Online Help	D 1 . 4
Other Important Keys	D 1 . 5



Starting, Moving Around, and Leaving

Advanced Revelation starts when you enter AREV at the operating system prompt. Use menus to move through the system. To leave Advanced Revelation, you can either interrupt your work temporarily (returning later to the same place) or exit completely.

Starting Advanced Revelation

To start Advanced Revelation, go to the directory that contains your Advanced Revelation files and enter AREV, like this:

```
C:\AREV>AREV
```

The Advanced Revelation copyright notice appears, followed by the Advanced Revelation Main menu:

```

Revelation Main Menu
Design Access SQL Tools Management Exit
"Design or modify data entry windows, reports, and menus."

```

When you want to:	Choose:
Design or modify data entry windows, reports, menus, popups, or macros	Design
Use existing windows and reports	Access
Use tools to maintain the database	Tools
Maintain accounts and system environment parameters	Management
Quit, go to operating system, or go to The Command Level (TCL)	Exit

Choosing Menu Options

The Main menu works like every menu in Advanced Revelation. To choose an option, highlight it and press [Enter] or [F9] (Save). Use any of these methods to move the highlight:

- To go directly to any option, type the first letter of the option you want
- To go to the next option in a list, press [Spacebar]
- To move through the options, use the arrow key that points in the direction you want to move
- To go to the first option, press [Home]
- To go to the last option, press [End]

A brief message across the bottom of the menu explains the purpose of the highlighted option.

The Status Lines

Two status lines at the bottom of the screen provide detailed information about where you are in Advanced Revelation and what kind of work you are currently performing. For more information about the status lines in a particular environment, see the “Using System Features” chapter.

Leaving Advanced Revelation

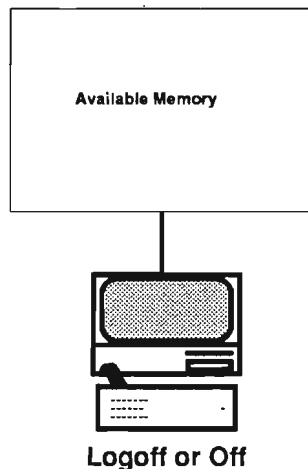
Advanced Revelation provides two ways to leave. If you are through working, you can quit Advanced Revelation altogether. You may also exit Advanced Revelation temporarily if you simply need to run one or more operating system commands.

You can leave Advanced Revelation either through the Exit menu or by entering the OFF command directly at The Command Level (TCL).

- To leave through the Exit menu, choose **Exit** from the Main menu.
- To enter a command at TCL, press [F5] (TCL) from anywhere in Advanced Revelation and enter the command on the command line. For information on TCL commands, see the *TCL* section.

When you are finished with Advanced Revelation and want to go on to other work, use the **Logoff** option on the Exit menu or the OFF command at TCL.

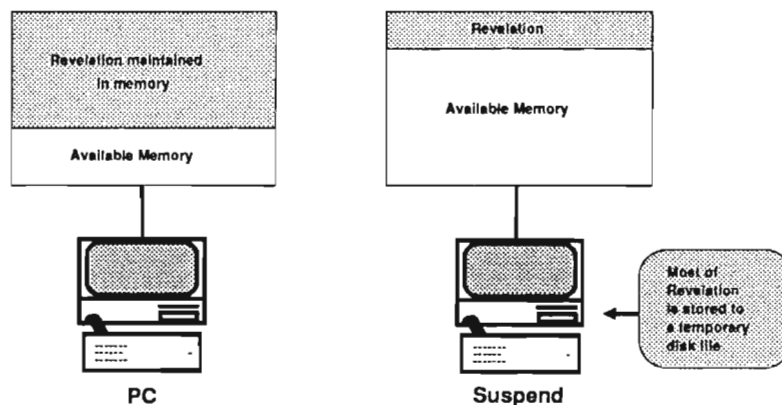
Logoff or OFF turns Advanced Revelation off.



Advanced Revelation provides two options for you to go to the operating system temporarily, use other programs, and then return to the same place in Advanced Revelation.

PC allows you to use MS-DOS commands and programs that require only a small amount of memory. The full Advanced Revelation system is retained in memory.

Suspend saves most of Advanced Revelation to a temporary disk file. Using **Suspend**, you can temporarily leave Advanced Revelation and use almost any command or program at MS-DOS.



With either option, enter **EXIT** at the operating system prompt to return to Advanced Revelation.

Getting Online Help

Help keys display information about your current task, options, and other Advanced Revelation functions.

Advanced Revelation provides six types of online help messages to assist you at any step while working in the system.

You can get specific help for your current task or you can go directly to a help message for any window, command, or function in Advanced Revelation. If you are not sure what to enter at a prompt, you can see all your options and choose what you want from a popup. In any context, you can see a list of active keys and an explanation of how to use them to make your job easier.

When you want to:	Press this key	
Read help text about the current prompt or popup	[F1]	Context
Read help text about the concept behind the current window or process	[Ctrl-F2]	Concept
Choose help for a window, command, or function from a menu	[Ctrl-F1]	General
View a list of all keys that are active currently	[Ctrl-F9]	Keys
Choose options for the current prompt or context from a popup	[F2]	Options
Choose Softkey functions for the current window from a popup	[F6]	Softkeys

For more information about online help, see the “Using System Features” chapter in this section.

Other Important Keys

In addition to the Help keys, the most frequently used keys in Advanced Revelation are [Esc], [F5] (TCL), [F9] (Save), and [F10] (Menu). Use these keys to accomplish tasks and move quickly to any part of the system.

When you want to:	Press this key:
Return to the previous window or menu	[Esc]
Save the information on the current window or choose the highlighted menu option	[F9]
Go to The Command Level (TCL)	[F5]
Go to the Main menu	[F10]

Advanced Revelation is designed for multilevel activity: you can interrupt one task, pick up another, then return to your original place with a few quick keystrokes.

In fact, you can layer a great many tasks in Advanced Revelation, directly accessing any part of the system from any other part. You are only limited by available computer memory.

To suspend the current activity and pursue another task, press [F5] to go to The Command Level (TCL) and then press [F10] (Menu) to enter the menu system. As you finish a task, press [Esc] to exit the current process. This will return you to the previous level, where you can either resume a suspended task or press [Esc] again to exit to the next level.



Chapter D 2

Using System Features

This chapter provides detailed information about Advanced Revelation's menu system and explains how to use menus, windows and popups as you build your application. It also explains how to control the online help system.

The Advanced Revelation Environment	D 2 . 1
Managing the Advanced Revelation Environment	D 2 . 2
The Online Help System	D 2 . 2
What the Status Line Tells You	D 2 . 4
Using Menus	D 2 . 6
Using Windows	D 2 . 7
Using Popups	D 2 . 18



The Advanced Revelation Environment

The Advanced Revelation environment provides you with a complete, well-organized set of tools for developing, maintaining, and using database applications. The various tools are used to organize the different aspects of your applications.

Because of Advanced Revelation's flexibility, you build your database and application using the approach most comfortable to you. You can create a quick prototype of your system, then refine the prototype until your design is complete. Or, you can come to Advanced Revelation with a finished design and build your system from the bottom up.

There are three major types of tools available in the Advanced Revelation environment: menus, windows, and popups. You use menus to move through Advanced Revelation, windows to issue commands and design applications, and popups for help in bringing information into the system. This section provides an overview of each element in the Advanced Revelation environment.

Menus in Advanced Revelation organize the tasks and options of an application into logical groups. For example, to design windows, reports, menus, popups, and macros, you use the Design menu. To use Advanced Revelation's various tools for maintaining your database, you use the Tools menu.

If menus are the paths to Advanced Revelation's work areas, windows are the media that get the job done. Use windows to issue commands, to specify your database and application definitions, and to manipulate data. Windows are also used to enter and display information in your database.

Popups provide support throughout the system. Like a staff of expert assistants, they inform you about your options in many situations, guide you to the correct response, and help you use the system efficiently.

If you prefer, you can accomplish the same goals reached through the menu system by issuing commands directly at The Command Level (TCL). TCL is a command line you access directly by pressing the TCL key [F5]. For more information on TCL, see the *TCL* section.

Managing the Advanced Revelation Environment

When you log on to Advanced Revelation, you have available a complete application environment. Many aspects of the environment can be modified to suit the needs and preferences of individual Advanced Revelation users.

The **Management-Environment** option from the Main menu calls the Environment menu. Options on this menu enable you to customize the standard environment settings provided in Advanced Revelation. Commonly changed environment settings include specifying whether to display the Main menu or the TCL command window when first logging on to Advanced Revelation, changing the help level, specifying a macro set, and turning Editor functions off and on.

- ☛ For more information about user environments, see the "Users & Accounts: Planning Access and Security" chapter in *Developing an Application*.

The Online Help System

One of the many features in Advanced Revelation is the ability to set a Help level. The Help level that you specify is used to determine the friendliness of the Advanced Revelation environment.

No matter what Help level you choose, the online Help system is immediately available. For an overview of the Advanced Revelation Help system, press [Ctrl-F1] (General). To obtain concept ("big picture") information about the current system window, just press [Ctrl-F2] (Concept). At popups or the current prompt, press [F1] (Context) for specific help about what to enter or select.

- ☛ If you set the Help level to 1 (Novice), a help message will automatically appear on the right side of the screen each time you enter a window in Advanced Revelation. You must clear the automatic help message before performing any other operation in the window. To clear any help message, press [Esc].

About the Help Levels

Your Help level controls whether help messages appear automatically, and dictates which version of the status line appears at the bottom of the screen.

The Help level can be set to the values 1, 2, or 0 (zero). The values correspond to your familiarity with Advanced Revelation. Setting the Help level to 1 (Novice) causes Advanced Revelation to present help messages automatically, and list active function keys in the status line. A Help level of 2 (Intermediate) provides a status line listing active function keys, but displays help messages only on demand. A Help level of 0 (Advanced) provides help messages only on demand, and displays the full status line.

- ☛ If the Help level is set to 1 (Novice), Advanced Revelation will always prompt you to select a Help level at login time. If the Help level is set to 0 or 2, you will no longer be prompted to set a Help level for the current session when logging in. If you want to change the Help level after this, you can do so by editing the General System Configuration window. See "Changing Your Help Level" below.

Automatic Help Messages

If the Help level is set to 1, a help message will appear automatically each time you enter a new window. This provides you with an introduction to the window, and gives reminders about active keystrokes and other specific information.

For example, if the Help level is set to 1 and you enter Paint, a help message will appear explaining the basics of Paint. As each new window in the Paint process is called (the Makefile window, the Customize window, etc.), a help message introduces that window.

You can page up and down in the help message using the [PgUp] and [PgDn] keys. When done reading the help message, you can press [Esc] to continue on to the window.

Status Line

The status line that appears at the bottom of the screen will also reflect the current Help level. The status line for Help levels 1 and 2 displays information about keystrokes active in the current window. Help level 0 (zero) displays a status line that substitutes detailed information about the current window for the keystroke summary.

For example, Help levels 1 and 2 display a status line similar to the following when in Paint Roam mode:

Paint	Roam	Col 2/78	Append mode	Auto	bind	CUSTOMER	F4-edit	F9-save
F1-Help CF2-Concept Help F2-Options F4-Edit F6-Softkeys F10-Menu Esc-Exit								

The same status line appears like this when the Help level is set to 0 (zero):

Paint	Roam	Col 2/78	Append mode	Auto	bind	CUSTOMER	F4-edit	F9-save
Ed Off		Row 2/20	Macro	User				Level 2

The status line for Help levels 1 and 2 displays less detailed information about the current window. Specifically, the following types of information are not noted in the status line for Help levels 1 and 2:

- Editor status
- Filter active
- Row designation (for Paint and editing multiline prompts)
- Current macro set
- User or account name
- Indexing active
- Window level

For more information about the status line, see "What the Status Line Tells You" in this chapter.

Changing Your Help Level

The Help level can be set at the General System Configuration window. To access this window, choose the **Management-Environment-General** option from the Advanced Revelation Main menu.

A prompt named *Help Level* appears at the top of the General System Configuration window. You can change the Help level at this prompt. Enter the desired Help level. You can press [F2] (Options) at the prompt to select the Help level from a popup of choices. Press [F9] to save the change and exit the General System Configuration window.

Press [Esc] to return to the Main menu. Now choose the **Exit-TCL** option from the Main menu. The Command Level (TCL) window will display. Enter the command:

```
RESET
```

This will reset the system and activate the new Help level.

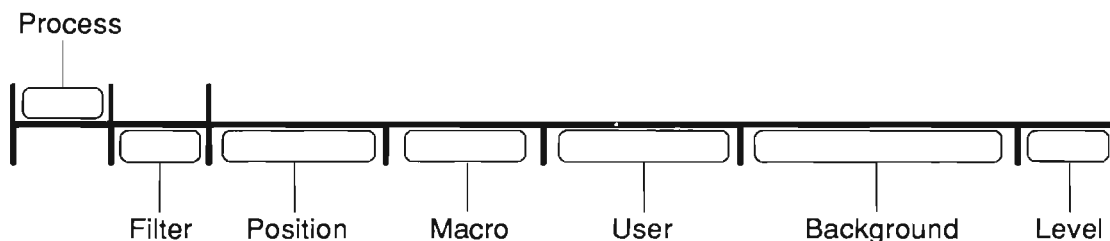
What the Status Line Tells You

The status line that appears at the bottom of the screen reflects the current Help level. The status line for Help levels 1 and 2 displays information about keystrokes active in the current window. Help level 0 (zero) displays a status line that provides detailed information about where you are in Advanced Revelation and what kind of work you are currently performing.

This section describes the information provided at the detailed status line that is displayed when using Help level 0.

Status Line Information

Some boxes in the status line display the same kind of information throughout Advanced Revelation. The illustration below shows these boxes. A description of each box follows the illustration.



- Process** The Process box names the current process. For example, if you are at a menu, this section of the status line displays "Menu." If you are at a window or popup, it displays "Window" or "Popup," respectively. If you are using the Editor, "Editor" displays.
- Filter** The Filter box shows if a filter is active. A filter contains criteria used to select some of the records in a file. When a filter is active, the records meeting a specified selection criterion are available for display in an application window or for use in some process. For more information on using filters, see the "Using the Filter Key" chapter in *Accessing Data*.
- Position** The Position box specifies the cursor's current row position and the total number of rows for the multivalued prompt, text prompt, or window. For example, "Row: 5/80" means that the current window has a total of 80 rows and the cursor is positioned on the fifth row.
- Macro** The Macro box names the set of macro keys active for the current user. Macro keys are predefined keystrokes that can carry out entire sequences of keystrokes or commands.
- User** The User box names the current user or account (the name under which you logged on).
- Background** The Background box provides information about background indexing. "Updating Indexes" is displayed when background indexing is active.

Level The Level box shows the current level number. When you first start Advanced Revelation, you are at Level 1. Each time you choose a new menu option or access a new window, your level number increases by one. Press [Esc] to return in order through the levels.

Using Menus

The menu system logically organizes Advanced Revelation processes and provides a pathway to each process. This section explains how to use Advanced Revelation menus.

Choosing Menu Options

A brief message across the bottom of the menu explains the purpose of a currently highlighted option. To choose a menu option, highlight the option you want and press [Enter] or [F9] (Save). Use any of these methods to move the highlight:

- To go directly to any option, type the first letter of the option you want. If more than one option begins with the same letter, each will be highlighted in turn.
- To go to the next option in a list, press [Spacebar].
- To move through the options, use the arrow key that points in the direction you want to move.
- To go to the first option, press [Home.].
- To go to the last option, press [End].

Finding the Menu You Want

You can display any menu in the system by choosing options through the menu structure, or you can go directly to any menu by using the Options key [F2].

To go directly from one menu to another, press [F2] at a menu. A popup appears, listing every menu in the system, as illustrated by the following:



The menu popup is a single-selection popup, which means you choose one option in the list and Advanced Revelation immediately responds. To choose an option, either type the number of the option you want and press [Enter] or highlight the option you want and press [Enter]. When you choose an option from this popup, the menu named by the option appears.

For more information on using popups, see "Using Popups" later in this chapter.

Using Windows

Use windows to issue commands, design an application, or modify data in a database.

- For information about how to move from prompt to prompt and how to edit data at a prompt see the "Entering and Editing Data in Windows" chapter in *Getting Started*.

What Are Windows?

Windows are the bordered areas on the screen that contain prompts and entry lines. There are two types of windows in Advanced Revelation: collector windows and application windows. Collector windows gather information for a process you are executing. Application windows are used to enter data into files.

Within Advanced Revelation, collector windows are used to issue commands and to provide Advanced Revelation with the information it needs to perform a process. For example, when you choose **Access** from the Main menu and then **Window** from the Access menu, Advanced Revelation displays a collector window:

Window Name
Enter a name for the window

Window Name is a collector window because it collects information to use in a process. You enter the name of the data entry window you want to use, and Advanced Revelation responds by displaying it.

In the applications that you create with Advanced Revelation, windows are used to enter, view, and modify data in your database. For example, you might design a Customer window to enter and modify data in the CUSTOMER file. Windows used to enter and modify data in your database are called application windows.

Prompts, Entry Lines, and Labels

Windows consist of several elements:

- *Prompts* identify the data to be entered or displayed
- *Entry lines* are the areas used to enter or display data
- *Labels* are text that is permanently displayed in the window but not associated with any prompt or entry line

Virtual Space

The complete layout of prompts and labels defined for a window is called the window's virtual space. In some cases, a window's virtual space is larger than the space contained within the window's borders. A broken border indicates more material in the border's direction.

For information on moving through a window's virtual space, see "Panning Through a Window's Layout" later in this chapter.

Saving Entries at Windows

Once you have entered data at a window, you can write your entries to the file (in an application window) or have Advanced Revelation respond to your instructions (in a collector window) by pressing [F9] (Save).

In a window that contains only one prompt, you can use either [F9] or [Enter] to save your entry. In windows that contain more than one prompt, [Enter] moves the cursor from prompt to prompt, while [F9] (Save) saves the window's current values.

To leave a window without saving the current entries or activating an Advanced Revelation process, press [Esc]. If you have changed values at the window during the current session, Advanced Revelation asks you to press [Esc] again to confirm that you do not want your changes saved.

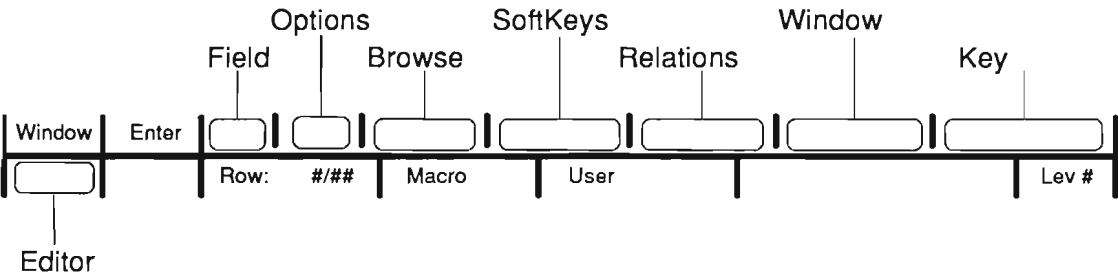
Refreshing Windows (F8)

Use [F8] (Refresh) to clear all the data displayed at the current window. In application windows, the data is cleared without updating the record. In collector windows, the data is cleared without initiating a process.

If you make an entry at any entry line on the screen, and then press [F8] (Refresh), Advanced Revelation asks you to press [F8] again to confirm that you want to abandon your changes. When you confirm, Advanced Revelation clears the data. You can then resume data entry.

The Status Line for Windows

When the Help level is set to 0, the status line at the bottom of the screen tracks screen activity. The following illustration shows the kinds of information displayed by the status line when you are using a window. A description of common sections of the status line follows the illustration.



Fields Fields are used to store the data you enter and edit in windows. The status line displays information about the field that is receiving data from the current entry line. The table below shows the kinds of fields:

Table D2-1
Kinds of Fields

When the Status Line displays:	The field is:	The type of data displayed is:
Sv	Single Value	A single value on one line
Txt	Text	A string of text that extends over one or more lines
Mv	Multivalue	A series of distinct values that extends over one or more lines
Amv	Associated	A series of distinct and Multivalue related values in separate multivalue prompts
Key	Key	The record's key or name, used to give a unique identity to each record in a data file
Dct	Dictionary	The name of the dictionary to use
FN	Filename	The name of the data file to use

- Options** The Options status displays when options are available for the current prompt. When options are available, you can press [F2] (Options) to display a popup of possible entries for the current prompt.
- Browse** The Browse status displays when a browse list is active. Browse lists are used to view and edit a specific set of records from a file.
- Softkeys** The Softkey status displays when softkeys for the current window are available. The application developer defines softkeys to run special processes for a window. For example, a softkey can be used to run a report from a window. Softkeys are any of the keys [Shift - F1] through [Shift - F10]. The [Alt - F1] through [Alt - F10] keys can be used as well, although they are normally defined as Relations keys (see below). When softkeys are available, you can press [F6] (Softkeys) to see and choose softkeys from a popup. For more information about softkeys, see "Using Softkeys" later in this chapter.
- Relations** The Relations status displays when windows related to the current window are available. Related windows are used to view and edit records in related files. When related windows are available, you can press [Ctrl - F6] (Relations) to see and choose related windows from a popup. Usually [Alt - F1] through [Alt - F10] are used as Relations keys to call related application windows. For more information on related windows, see "Displaying Related Windows" later in this chapter.
- Window** If a window is collecting information to run a program, the status line displays "Collector". Otherwise, it shows the name of the active file or the name of the active window.
- Key** The Key status displays the key to the current record. The key is used to distinguish one record from another.

Accessing Other Windows or Menus

While you are using one window, you can suspend your task and go directly to other windows or menus, then return to your original window and resume work. To access other windows or menus, use [F10] (Menu), [F5] (TCL), [F6] (Softkeys), or [Ctrl-F6] (Relations).

- Using the Menu Key** Press [F10] (Menu) to display the Main Menu. Choose the appropriate menu options to perform whatever tasks you wish. When you are finished, press [Esc] to return through the levels to your original window.

Using The Command Level (TCL) Key

Press [F5] (TCL) to display The Command Level window. You can display any window and accomplish any Advanced Revelation task by issuing commands directly at The Command Level (TCL).

Displaying Related Windows

Related windows allow you to view and edit records in related files. Files are related when a field in one file contains record keys for another file. For example, invoices in an INVOICE file might be related to the appropriate customer record in the CUSTOMER file through a joined field called INV_KEYS in the CUSTOMER file. The INV_KEYS field holds the keys to invoice records in the INVOICE file.

When a related window has been defined for the current window, the status line displays "Relations". Press the Relations key [Ctrl-F6] to view a popup of all the windows related to the current window. The popup should look similar to this:

Related Application Windows					
	Key	Window	File	Joined Field(s)	Intersections
1>	AF1	MEMBERS	TEMPLATES	INV. KEYS	none
2	AF2	INVOICES	TEMPLATES		3

To access a related window, highlight your choice and press [Enter], or type the number of your choice and press [Enter].

To return to your original window, press [Esc].

The Related Application Windows popup displays information about the related windows under these headings:

Key The Key column shows the Relations key that has been defined to display the related window. To display the related window directly without going through the popup, press the Relations key from your application window. Usually [Alt - F1] through [Alt - F10] are set up as Relations keys used to call specific related application windows.

Window The Window column names the related window.

File The File column names the file where the related window template is stored.

Joined Field(s) The Joined Field(s) column names the field or fields in your current file that contain record keys for the related file.

Intersections The Intersections column indicates the number of record keys in the joined field (for the currently displayed record). For example, if the joined field INV.KEYS in a customer record in the CUSTOMER file contained five invoice numbers, the Intersections column would show five intersections.

If you choose a window with intersections, the Joined Key(s) popup appears, listing each of the values in the joined field. Use this popup to create a browse list of the records you want to see. One at a time, highlight the record keys you want and press [Enter] or type the number next to each of the record keys you want and press [Enter]. When you are finished choosing record keys from the popup, press [F9] (Save) to save the list and call the related window. The data for the first record in the browse list will appear in the related window.

For more information about using browse lists, see the "Selecting and Browsing through Data" chapter in *Accessing Data*.

Using Windows for Reporting

Windows in Advanced Revelation can play a central role in the reporting process. You can take advantage of the Query or index search capabilities to look up information in the database. Records containing the information that you want are listed on a browse list. Once a browse list is active, it can be used in the window to view the records. For more information on using browse lists, see the "Selecting and Browsing through Data" chapter in *Accessing Data*.

Query in Windows (\) Any application window can become a form for creating a database search by using Query. Query is invoked in a window when you press \ (backslash) at the key prompt, which is usually the first prompt in the window.

Once Query is invoked, you enter selection and sort criteria at the prompts in the application window in order to create a search. When you execute the search, records that meet the search criteria are returned to the application window as a browse list.

For more information about the Query process, see the "Using Query" chapter in *Accessing Data*.

Index Search Options

If you have established indexes for your data files, you can make use of these indexes when looking up data in windows.

To perform a direct index search, you can use \ (backslash) plus a search value directly at the key prompt.

You can search indexes not just for exact matches of data, but for substrings, ranges, and specific patterns. The Browse Select window, accessible from the Filter key [Ctrl-F10], prompts you for information used to search the indexes.

The index search capabilities are described in detail in the "Using Indexes" chapter in *Accessing Data*.

Creating an R/LIST Report (Alt-V)

A keystroke available in application windows enables you to create and execute R/LIST reports directly from within the window. If a browse list is active, the records in the browse list are passed to the R/LIST command as a select list (filter).

Press [Alt-V] to display the R/LIST window. Enter an R/LIST command into this window, and press [F9] to execute the command. Press [Esc] to return to the window without executing an R/LIST command.

When the report is finished, your application window will be redisplayed. If you had a browse list active when you created the report, the browse list will remain intact.

For more information about the [Alt-V] key see the "Creating Reports with Advanced Revelation" chapter in the *Creating Reports* section. For more information on creating R/LIST reports, see the chapters called "Creating R/LIST Reports" and "Writing Reports with EasyWriter" in the *Creating Reports* section. Also see "Introduction to R/LIST" in the *R/LIST* section.

Moving a Menu, Window, or Popup (Ctrl-F8)

You can temporarily move a menu, window, or popup to a more convenient location on the screen. The menu, window, or popup can move until the border touches the edge of the monitor or the top of the status line.

To move a menu, window, or popup, press [Ctrl-F8] (Move). Use the arrow keys to move the menu, window, or popup. When you are finished, press [Enter] or [Ctrl-F8] to turn Move off.

Panning Through a Window's Layout (F7)

Sometimes a window's virtual space is larger than the area displayed within the window's borders. A broken border line indicates more material in the border's direction. Panning allows you to move through virtual space in a window so that you can see the contents that are outside the current window borders. As you pan, the window's entire frame is shifted by a specified increment (a step, a window width, a page).

To pan through a window display without changing the size of the window's borders, press [F7] (Pan). Use the arrow keys to pan through the layout. When you are finished, press [Enter] or [F7] to turn Pan off. You return to the prompt from which you started Pan.

Use these keys in Pan to pan through the entire layout:

When you want to Pan:	Press this key:
Up one step	[↑]
Up one page	[PgUp]
Down one step	[↓]
Down one page	[PgDn]
Right one step	[→]
Right one window width	[Ctrl →]
Left one step	[←]
Left one window width	[Ctrl ←]

Interpreting the Status Line

When you press [F7] (Pan), and as you pan through the window, the status line displays information about the current location of the window borders in relationship to the defined edge of virtual space for this window.

Col #/#

The number to the left of the slash (/) shows the number of columns that the window has been panned to the right. The number to the right of the slash shows the maximum number of columns that the window can be panned to the right. For example, Col 5/25 means that the entire window frame has already moved five columns to the right (so five columns on the left are hidden) and that it can be moved a maximum of 25 columns right. Thus, when Col 25/25 displays on the status line, the window frame has panned the maximum number of columns that it can move to the right.

Row#/# The number to the left of the slash shows the number of rows that the window has been panned down. The number to the right of the slash shows the maximum number of rows that the window can be panned down. For example, Row 0/14 means that the entire window frame can be moved down a maximum of 14 rows, but that it has not moved yet. Thus, when Row 14/14 displays on the status line, the window frame has panned the maximum number of rows that it can move down.

☛ If you are using Help level 1 or 2, Row #/# will not display at the status line.

You can change the size of steps that the entire window frame moves at each press of an arrow key. The current step sizes are displayed on the status line. For more information, see "Changing the Size of Steps" in this chapter.

Resizing a Window (Ctrl-F7)

The [Ctrl-F7] (Resize) key allows you to change the border size of the currently displayed window.

Use Resize at an existing window to make room for other material on the screen or to see more of the window's prompts at one time. When the virtual space allocated to a window is larger than the area the window occupies on the screen, a broken border line indicates that there is more material in the direction of that border. You can increase the size of a window in the direction of a right or bottom broken border. You can Resize a window to be as small as one column by two rows, or as large as the virtual space defined for this window, up to one full screen.

To change the size of the current window, press [Ctrl - F7] (Resize). Use the arrow keys to push the bottom and right borders in the direction you want. When you are finished resizing, press [Enter] or [Ctrl - F7] to turn Resize off.

Use these keys in **Resize** to resize the window's right and bottom borders:

When you want to move the bottom border:	Press this key:
Up one step	[↑]
Up one half window depth	[PgUp]
Down one step	[↓]
Down one half window depth	[PgDn]

When you want to move the right border:	Press this key:
Right one step	[→]
Right one half window width	[Ctrl →]
Left one step	[←]
Left one half window width	[Ctrl ←]

Interpreting the Status Line When you press [Ctrl F7] (**Resize**), and as you resize the window, the status line displays information about the current location of the window borders.

Col 0/# The first number is always zero. The second number indicates the relationship of the right border to the edge of the window's virtual space. A zero to the right of the slash indicates that the right border is at the edge of the window's virtual space. Any number to the right of the slash indicates how many columns the right border is away from the edge.

In other words, the second number shows the maximum number of columns that the right border can move right. For example, *Col 0/16* means that the right border can move up to 16 columns to the right (that it is 16 columns in from the edge of virtual space). Thus, when *Col 0/0* displays on the status line, the right border has moved the maximum number of columns that it can move to the right. The border has reached the edge of the window's virtual space.

Row 0/# The first number is always zero. The number to the right of the slash shows the maximum number of rows that the bottom border can move down before reaching the edge of the window's virtual space. For example, *Row 0/14* means that the bottom border can be moved down a maximum of 14 rows. Thus, when *Row 0/0* displays on the status line, the bottom border has moved the maximum number of rows that it can move down.

- ✎ If you are using Help level 1 or 2, Row 0/# will not display at the status line.

Changing the Size of Steps

A step is the number of columns or rows that the entire window frame will move (in Pan mode), or that a window border will expand or contract (in Resize mode) each time an arrow key is pressed. The current step sizes are displayed on the status line.

Whenever Pan or Resize is on, you can change the size of the step used by the Resize and Pan keys. Use these keys to change the size of steps:

When you want to:	Press this key:
Increase row step by one	[Tab]
Decrease row step by one	[Shift -Tab]
Increase column step by one	[Spacebar]
Decrease column step by one	[Backspace]

Interpreting the Status Line When you press [Ctrl - F7] (Resize) or [F7] (Pan), the status line displays information about the current size of the steps used to resize or pan:

Window	Resize	Col 1/11	by 5 Cols	by 2 Rows	<Enter>-end <esc>-exit
Ed Off		Row 0/5	User		Level 1

by # Cols The status line shows the number of columns that the border will move (Resize) or that the entire window frame will shift (Pan) at each press of the arrow key (each step).

by # Rows The status line shows the number of rows that the border will move (Resize) or that the entire window frame will shift (Pan) at each press of the arrow key (each step).

Using Softkeys

Softkeys execute special functions. Each window in Advanced Revelation can have its own set of softkeys.

The softkeys provided in a window usually help you to perform activities related to that window. For example, a softkey might run a report, call another window, call a special menu, or call a related program.

Softkeys are any of the keys [Shift-F1] through [Shift-F10] or [Alt-F1] through [Alt-F10]. At the time that a window is designed, specific softkeys may be defined for that window.

To display a popup that lists the softkeys defined for your current window, press [F6] (Softkeys). To execute a softkey from the popup, highlight the appropriate process on the list and press [Enter] or type the number of the appropriate process and press [Enter].

You can execute a softkey directly by pressing the appropriate keys. For example, suppose the softkey [Shift-F1] were defined to run a report. Either of these methods will run the report:

- Press [Shift-F1].
- Press [F6] to display the Softkey popup and choose [Shift-F1] from the popup.

Using Popups

A popup displays information or lists options related to the current activity. It returns selected values to the location from which it was called, or performs an action such as a softkey action.

There are several types of popups:

- Single-selection popups perform a procedure or write a value to a single-value field
- Multiple-selection popups create a list of fields, records, or files, or write values to a multivalue field
- Display-only popups display system information or help messages

Anytime a popup is displayed, you can exit from it without making any selections by pressing [Esc].

Interpreting the Status Line

When a popup appears, the status line displays Popup in the first box. The second box describes the type of popup you are using. The most common types are single-selection (Single), multiple-selection (Multi), and display-only (Display).

The status line also displays a message that explains how to make selections from the current popup.

Moving the Highlight

One way to select an item from a popup is to highlight your choice. To move the highlight to the item you want, use the arrow keys.

If a popup contains more items than it can show on one page, a message at the bottom border shows the current page number and the total number of pages. To move from page to page of a popup that contains more than one page, use [PgUp] and [PgDn].

Highlighting Options by Alphabetic Character

Popups allow you to highlight an item by typing an alphabetic character. This method can be used in addition to typing a number or using the cursor keys to highlight an item in a popup window.

To highlight an item by alphabetic character, simply type the alpha character that begins the item text. For example, to highlight the item "Convert to uppercase", you would type C. When you type any alphabetic character while in a popup, a message will appear at the bottom of the screen that reads:

Text to find:

The character you typed will display next to this message.

Press [Enter], or add text and then press [Enter]. The cursor will move to the next selection in the popup in which the first column begins with the character or string you have entered.

If the character or string you have entered does not appear in the first column of the popup, a message appears alerting you to this fact.

You can also use the [Ctrl-F] (Find) key to find any string in a popup. Press [Ctrl-F], and at the *Text to find* prompt enter a string of characters that appears in the popup. If the string is found in any column of the popup, the highlight bar will move to that row.

Choosing Items from Single-Selection Popups

Use either of these methods to choose an item from a single-selection popup:

- Highlight the item and press [Enter] or [F9]
- Type the number of the item and press [Enter].

Choosing Items from Multiple-Selection Popups

To choose items from a multiple-selection popup, highlight the item you want and press [Enter] or type the number of the item and press [Enter]. The selected item will blink or otherwise changes its video attribute. Choose additional items the same way.

If you change your mind about an item, move the highlight to the item and press [Enter] again. The [Enter] key works as a toggle for choosing and rejecting items.

When you have chosen all the items you want, press [F9]. Advanced Revelation returns you to the process from which you called the popup, bringing your selections from the popup back to the process.

For example, if you press [F2] (Options) at the *Source record(s)* prompt in the Recordcopy window, Advanced Revelation displays a multiple-selection popup listing all the available records. You can choose as many records as you want from the popup. When you return to the Recordcopy window, the chosen records will be entered in the *Source record(s)* prompt.

Using Display Popups

Display popups display information or help about your current activity. For example, the List Files popup displays a list of all attached files. To see the popup, choose List Files from the Files Tools menu, or enter the command LISTFILES at TCL. You can page through the list of files, but you may not make selections from this popup. To exit the popup, use [Esc].

Chapter D 3

Entering and Editing Data in Windows

This chapter describes the protocol of using windows in Advanced Revelation. Included in this chapter are descriptions of how to move from prompt to prompt in a window, as well as information about editing data at a prompt.

Moving the Cursor D 3 . 1

Entering and Editing Data in Windows D 3 . 6



Moving the Cursor

When using a window, you can move forward and backward within a prompt or from prompt to prompt. You can also make use of tab stops or a prompt popup to move quickly across several prompts. If the window consists of several pages in virtual space, you may be able to use the page keys to move from page to page.

Editor Status

The use of cursor keys at multilined prompts (text or multivalued prompts) is affected by the status of the Editor. The Editor can be on or off. If the Editor is on, cursor movement will be within the prompt. If the Editor is off, the cursor will move between different prompts in the window.

To toggle the Editor on or off, use the [F4] key. The status of the Editor is displayed at the status line. For specifics about the effect of the Editor mode on cursor movement keys, note the entries in the following tables carefully.

When entering or editing data at a prompt, you will be using a full featured text editor. The Editor permits you to edit individual characters, words or lines. In addition, you will be able to use features of the Editor to cut and paste blocks of text at the current prompt entry line, or search for specific text within the current prompt entry line.

As you work with windows, remember that [Ctrl-F9] will provide you with a summary of keystrokes which are active.

The tables that follow explain how to move the cursor within a prompt's entry lines or from prompt to prompt.

Because of the various types of prompts that are available in Advanced Revelation windows, there are separate instructions for moving within and between text, multivalued and associated multivalued prompts.

Moving Within a Prompt Entry Line

The following table summarizes how to move the cursor within the entry line for a prompt. Note that the keys used for vertical movement ([↓], [PgUp], etc.) are active at prompts which contain more than one line of data (text and multivalued prompts).

Moving:	Press this key:
Right one character (non-destructive)	[→]
Right one character (destructive)	[Spacebar]
Left one character (non-destructive)	[←]
Left one character (destructive)	[Backspace]
Right to the next tab stop	[Tab]
Left to the previous tab stop	[Shift-Tab]
Right one word	[Ctrl→]
Left one word	[Ctrl←]
To the last character in the line	[End]
To the first character in the line	[Home]

At text prompts and at multivalue prompts containing data (Editor on)

Down one line	[↓] or [Enter]
Up one line	[↑]
Down one page of data in a multivalue prompt	[PgDn]
Up one page of data in a multivalue prompt	[PgUp]

Moving from Prompt to Prompt

The following table summarizes how to move between different prompts in a window. When moving from or to a prompt containing more than one line (text or multivalue prompts), note that the meaning of certain keys ([↓], [PgUp], etc.) is dependent on the status of the Editor.

Condition	Moving:	Press this to key:
From a single-value prompt, or the top line of a multivalued prompt or text prompt	Up one prompt	[↑] or [Ctrl-PgUp]
From a single-value prompt, or an empty bottom line of a multivalued prompt or text prompt	Down one prompt	[↓], [Enter], or [Ctrl-PgDn]
In all cases	To the record's key prompt	[Alt-K]
	Forward one tab prompt	[Alt-T]
	Backward one tab prompt	[Alt-Y]
	To the middle prompt	[Alt-W]
	To a prompt you choose from a popup of all available prompts	[Alt-A]

☞ In order to use the tab prompts, the window must previously have had tab settings established.

For example, if you choose Management from the Main menu and Set Video from the Management menu, the Set Video window appears:

Set Video	
Application Windows	
Border	N
Title	N
Background	N
Prompt	N
Label	N
Entries	N
Highlight	
Border type	3
Popup Windows	
Border	N
Title	N
Background	N
Bar	N
Selections	N
Text	N
Border type	2

The data in this window sets the video display of menus, windows, and popups in your system.

When the window appears, the cursor is on the entry line for the first prompt. To move to the next prompt, use [↓], [Enter], or [Ctrl-PgDn]. To move to the previous prompt, use [↑] or [Ctrl-PgUp].

Some prompts are marked with a tab stop. To skip to the next tab prompt, press [Alt-T]. To skip to the previous tab prompt, press [Alt-Y].

The [Alt-A] keys provide a convenient way to go directly to the prompt you want. When you press [Alt-A], a popup appears listing all the window's prompts. To choose a prompt, move the highlight to the one you want and press [Enter].

Protected Prompts

Application developers may protect the data displayed at some of the prompts in a window. If a prompt has been designated as a protected prompt, the cursor will not stop at the prompt. Instead, the cursor will skip right over the prompt to the next prompt that can be edited. If a prompt is designated as filled and protected (FP), the window will automatically fill in the default data as the cursor skips over it. If a prompt is designated as required and protected (RP), the window will prompt for data in new records, and protect the data from then on.

Moving in Multivalue and Text Prompts

When the cursor first moves to an empty multivalue prompt, the Editor is off. If you make no entry, [↓] or [Enter] sends you to the next prompt. As soon as you begin typing, the Editor turns on and [↓] or [Enter] sends you to the next multivalue entry line within the prompt.

To move to the next prompt from a multivalue prompt, press [↓] or [Enter] at the lowest empty entry line associated with the multivalue prompt. To move to the next prompt from a text prompt, press [↓] or [Enter] at a blank line at the bottom of the prompt entry area.

To move to a previous prompt from a multivalue or a text prompt, press [↑] on the top line.

You can jump directly to any specific value (line) in a multivalued or text prompt by using [Ctrl-G].

When you press [Ctrl-G] at multivalued or text prompt, or at an associated multivalued group, a message at the bottom of the screen asks:

Line number to go to?

Enter the value, indicating the line to which you wish to jump.

If you enter a number higher than the number of values in the prompt or associated group of prompts, you will be positioned at the last value in the prompt or group.

To enter a new value at a multivalued prompt, press [F4] to turn on the Editor and then press [Return] to move through the existing values to a blank entry line.

Moving in Associated Multivalue Prompts

Associated multivalue prompts contain data in a set of associated prompts, each of which is a multivalue prompt. A master prompt, always the first prompt in the set, is used to identify each set of associated values.

Associated multivalue prompts work in much the same way as any other kind of prompt. The following table explains keys with special behaviors in associated multivalue prompts:

When you want to:	Press this key:
Move up one row, staying in the same column	[↑]
Move down one row, staying in the same column	[↓]
Move to the next prompt in a row, or from the last prompt in a row to the first column in the next row	[Enter]
<i>With the Editor off:</i>	
Cycle through prompts in a row that contains data	[→] or [←]
From a blank bottom row, leave the area of associated prompts	[→], [↓], or [Enter]
From the first prompt, leave the area of associated prompts	[←] or [↑]

Moving through Virtual Space

In some cases, the complete layout of prompts and entry lines contained in a window is larger than the space visible within the window's borders. When this occurs, a broken border indicates there is more material in the direction of the broken border.

To move from the last prompt on a page to the next page, use any key that moves you to the next prompt, such as [↓] or [Enter].

Moving using Page Tabs

If the window is larger than the displayed area on the screen, one or more page tabs may have been set when the window was designed in order to make it faster for the user to display the hidden areas of the screen. When page tabs are set, you can move from page to page (page tab to page tab) in a window. To do so, use the [PgUp] and [PgDn] keys.

Moving from page to page operates in a circular fashion. You can move consecutively through the pages, going forward or back. If you page down at the last page, you circle back to the beginning. Similarly, from the first page you can page backward to the last page.

Entering and Editing Data in Windows

When you enter and edit data in Advanced Revelation windows, you are using a full-featured text editor. This section describes how to use various features of the Editor while at a prompt.

Advanced Revelation's Editor will enable you to add, change, and delete characters, words, and lines at any prompt in a window. Additional features of the Editor will enable you to create and manipulate blocks of text, and to search for data.

- ☞ The information presented in this section addresses the use of the Editor within any one prompt in a window. Although you will have access to all the features of the Editor, you will find that not all of these features are applicable to editing a single prompt. Furthermore, you should remember that the editing operations you perform (deleting, marking blocks, searching for text) are happening only within the current prompt (unless otherwise noted).

As you work with windows, remember that [Ctrl-F9] will provide you with a summary of keystrokes which are active.

Highlighting Changeable Prompts

A setting in the Set Video window allows you to designate a video attribute that is used to display prompts at which a user can enter or edit data. This setting enables you to create a visual marker for prompts used to enter and edit data, making them readily identifiable to a user.

Prompts that typically cannot be edited are those for symbolic fields and those that have been marked as protected. These prompts will not be highlighted when the window is displayed.

To make use of prompt highlighting, you will need to set the appropriate video characteristic in the Set Video window. Edit the *Highlight* prompt under the label Application Windows. Use [F2] to display a popup of video attributes from which you can choose. Save your changes in the Set Video window by pressing [F9].

When the highlight video attribute is established, prompt entry lines containing data that can be edited will display with this video characteristic. If custom video characteristics have been established for a window or a specific prompt, these characteristics will override the highlight video attribute.

Display Length versus Field Length

All fields in Advanced Revelation are variable length fields. If your entry is longer than the entry line provided by the window, the characters scroll across the entry line as you type them. Use the [→] and [←] keys or the [Home] and [End] keys to move through the characters. For details on moving the cursor in an entry line, see "Moving the Cursor" earlier in this chapter.

Destructive Editing at Prompts

You can choose whether prompt editing is destructive or non-destructive. The prompt *Destructive Edit Toggle* in the Editor Environment window determines how prompt editing will behave. Select **Management-Environment-Editor** from the Advanced Revelation Main menu to access the Editor Environment window. If you enter Y at the *Destructive Edit Toggle* prompt, data will be cleared from a prompt as you begin to type. If you enter N, existing data will be preserved.

If prompt editing is destructive, the existing data at a prompt edit line is cleared when you begin typing at the beginning of that prompt. It will be non-destructive only if you press [F4] (Edit) prior to editing the prompt.

In non-destructive mode, existing data is preserved. If you are using Insert line mode (a Y entry at the *Insert line on (Y/N)* prompt), anything you type is inserted in front of the existing data. If you are not using Insert line mode (an N entry at the *Insert line on (Y/N)* prompt), anything you type will overwrite the existing data.

Using the Edit Key (F4)

At prompts that provide more than one entry line, such as text, multivalue, and associated multivalue prompts, the action of certain direction keys changes depending on whether the Editor is on or off. For details on using direction keys in data entry lines, see the discussion "Moving the Cursor" earlier in this chapter. The lower left corner of the status lines shows you the status of the Editor.

When you move to a new entry line, the cursor displays as a block. This indicates that the Editor is off. As soon as you begin typing, the cursor changes to a line, indicating that the Editor is on. Use the Edit key [F4] to toggle the Editor on and off.

If you are operating in destructive edit mode and want to edit existing values at a new entry line, turn the Editor on before you start to type. If you start typing on a line that contains existing data when the Editor is off, the existing data disappears. To recover the original information, press [F4] before moving to the next line.

- ☛ Within windows, an Editor operation only affects the entry lines associated with the prompt at which the cursor is positioned. The Editor turns off automatically each time the cursor moves to the entry line for a new prompt.

Editing Characters and Words

You can use standard editing keys ([Del], [Backspace], etc.) to edit characters within a prompt. In addition, Advanced Revelation has defined special keys for editing entire words.

When editing characters, you can be in either Overwrite or Insert mode. When Overwrite is on, typed characters replace existing characters at the cursor. When Insert is on, typed characters are inserted at the cursor. Use [Ins] to toggle between the two. The box in the lower left corner of the status line shows whether Overwrite (Ed Ovr) or Insert (Ed Ins) is on.

*Table D3-1***Determining Editor Status**

When the Status Lines display:	This is the Editor's status:
Ed Off	The Editor is off
Ed Ins	Typed characters are inserted at the cursor position
Ed Ovr	Typed characters overwrite the characters at the cursor position

Use [Backspace] to delete the character to the left of the cursor. Use [Del] to delete the character on which the cursor is resting. To delete the entire word to the right of the cursor, use [Ctrl-Y].

[Ctrl-P] duplicates the previously typed character. This feature is convenient when you want to type a line using a graphics character (a high ASCII value character). For example, you may want to divide two paragraphs at a text prompt with a double-line border (ASCII 205). Type the character once ([Alt] + 205 on the number pad), then press [Ctrl-P] to repeat it. Hold down [Ctrl-P] for a continuous line of the character.

Editing Lines

The following table summarizes operations you can perform for an entire line of data.

When you are at a single-value prompt, there is only one line. In that case, operations you perform will affect the entire contents of the field. Certain line editing features (for example, [Ctrl-N] to insert a new line) will not be applicable.

At prompts containing multiple lines (text and multivalued prompts), line-oriented operations only affect one line or value at the prompt.

When you want to:	Press this key:
Delete all characters from the cursor to the end of the current line	[Ctrl-L]
Delete all characters from start of the current line to the cursor	[Ctrl-K]
Delete all characters in the current line	[Ctrl-X]

The following do not apply to single-value prompts:

When you want to:	Press this key
Delete the current line; move next line up	[Ctrl-D]
Insert a blank line; move current line down	[Ctrl-N]
At the cursor position, cut the current line into two lines	[Ctrl-C]
Join the current line with the following line	[Ctrl-J]

Copying a Previous Record

While in an application window, you can copy data from the record displayed previously. You can copy selected fields, or the record in its entirety.

Advanced Revelation's application windows remember the record you last displayed. Once you have displayed a record in an application window, you can save or clear the record, and then use special keystrokes to copy information from that record into the current window. Depending on which key you press, you can fill in any single prompt, or copy the entire previous record.

Advanced Revelation only remembers the previously displayed record until you exit the window by pressing [Esc]. The next time you enter the window, you will have to display a record first before there will be anything to copy.

Copying Selected Fields If you press [Alt-O] while at a prompt, the data displayed at that prompt from the previous record will be copied to the current prompt. For example, if the *City* prompt from the previous record contained the name "Chicago," you could fill in "Chicago" at the *City* prompt of the current record by simply pressing [Alt-O].

Copying a Full Record You can use [Alt-C] to copy the entire previous record to the current window. Enter the key at the empty window, then press [Alt-C]. The last record displayed will be copied to the current window. You can then edit any prompt normally.

The [Alt-C] keystroke copies every field of the previous record (including those not displayed in the window) except the key. You must provide the new key before pressing [Alt-C].

Deleting a Record

To delete the current record, press [Alt-D]. Before deleting a record, Advanced Revelation asks you for confirmation. To delete the record, press [Alt-D] again. To cancel the delete action and preserve the record, press any other key instead of [Alt-D].

Using the Line Modes at Text Prompts

When entering or editing data at a text prompt, you can change certain settings of the Editor to help you. The settings which you can alter are:

- | | |
|--------------------|---|
| Insert Mode | If insert mode is on, the Editor will insert characters within existing text rather than overwriting them. |
| Wrap Line | If this is toggled to on, as the data is being entered, the Editor will split lines which are too wide and continue them on the next line. The point at which the line will wrap is determined by the width of the prompt entry line. |
| Insert Line | If this is set to on, the Editor will insert a blank line each time [Enter] is pressed. |
| Indent Line | If this is set to on, the Editor will indent new lines to match the indentation of the previous line. |

To change or view the different modes, press [Ctrl-W] while editing at a text prompt. This will produce a popup with the status of each mode. To toggle the modes, move the highlight to the appropriate option and press [Enter]. Press [F9] (Save) when you are finished setting modes, or [Esc] to abandon any changes.

Expanding the Editing Area for a Multilined Prompt

To create a larger editing area for a prompt that displays more than one line, press the Zoom key [F3]. The editing area for the current prompt will expand to fill one half the screen. This will enable you to see more of your data while editing.

To save your changes and go to the next prompt, press [F9] and then [Esc].

Zoom is commonly used when there are not enough entry lines for all the data at a multivalued or text prompt. For example, a Customer window might include a multivalued *ADDRESS* prompt that displays two prompt entry lines. Suppose the address for the current record contains three lines:

```
The Tyler Building
1234 George Road
Suite 567
```

When the record displays on the window, only two lines of the address are visible. To see all three lines at once and edit them, press [F3] when *Address* is the current prompt. The prompt expands to display all of the values in the field.

When using **Zoom** with multivalue prompts, place one value on each line. When using **Zoom** with text prompts, you can format the data into as many lines as you wish.

- Pressing [F3] (**Zoom**) in a window places you in the full-screen Editor. All functions and keystrokes defined in the Editor will be applicable.

Using Defined Blocks of Text

You can define a block of text at a prompt and use that block in a variety of ways. You can copy it once or many times at any cursor position, or you can move or delete the block.

In addition, you can save blocks of text out of a prompt to a holding area called a "buffer" for future use. This feature enables you to create blocks of text that can be used at many different points without having to re-enter the text.

- In an application window, a defined block can comprise some or all of the entry lines associated with a single prompt.

For details on defining, copying, moving, deleting and saving blocks of text, see the following chapter.

Searching and Replacing

While working at a prompt, you can search for specific text that appears in that prompt. In addition, you can replace some or all appearances of specific text at the prompt with new text. This feature is explained in the next chapter.

Setting a Sequential Counter

At key prompts that use a sequential counter as the default value, you can reset the counter directly at the prompt.

To set the value of the counter, type = (equal) at the key prompt using a sequential counter default. A window will display asking you to enter a new default value. Enter a new value. After you save the new value, you will be reprompted to enter a value at the key prompt that uses the sequential counter.

For example, if you wish to set the default value of a sequential counter to 100, you would type = (equal) at the key prompt that uses the counter as a default value. A window will display asking you to enter a new value. Enter 100. The default value will be reset to 100, and you will be prompted again to enter data at that prompt.



Chapter D4

Using the Full-Screen Editor

The Advanced Revelation full-screen Editor is used to create, display, modify, and delete Revelation records and operating system formatted files. The chapter presents its features and explains how to use it effectively.

Introduction to the Full-Screen Editor	D 4 . 1
Using Defined Blocks of Text	D 4 . 5
Searching and Replacing	D 4 . 8
Using Value Windows	D 4 . 9
Using Softkeys in the Full-Screen Editor	D 4 . 10



Introduction to the Full-screen Editor

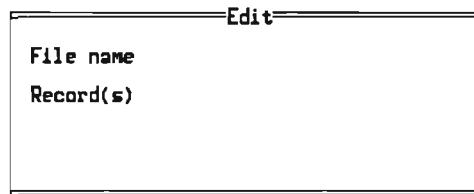
Use Advanced Revelation's full-screen Editor to enter or edit records in Revelation files. For example, you can use the Editor to edit R/BASIC source code records. You can also use Advanced Revelation's Editor to enter or edit operating system formatted files.

You use the same keys to enter and edit data in the full-screen Editor that you use in windows. For example, the same keys are used to move the cursor, enter or delete characters, enter or delete lines, etc. For details on the editing keys, see the "Entering and Editing Data in Windows" chapter.

Starting the Full-screen Editor

To start the full-screen Editor, choose **Edit** from the Developer Tools menu or type **EDIT** at The Command Level (TCL) window.

The Edit command window appears:



Edit	
File name	
Record(s)	

File Name

Enter the name of the file that contains records you want to edit, or the name of the file where new records you create will be written. The file name you enter must refer to an existing file. To choose from a list of attached files, press [F2] (Options).

Record(s)

Enter the name of the records to edit or create. To edit more than one record, enter the name of each record on a new line. To edit all the records in the file, enter * (asterisk). To choose from a list of available records, press [F2] (Options). The maximum size of a record you can edit in the full-screen Editor is 65,536 characters.

When you have entered the file name and record names you want, press [F9] to display the full-screen Editor.

You can also start the Editor by using the EDIT command at TCL. This is the general format for the EDIT command:

```
EDIT {DICT} filename record.key(s)
```

For more information on the EDIT command, see the "TCL Command Reference" chapter in the *TCL* section.

Setting the Editing Environment

You can determine these environmental features of the Editor:

- Case insensitivity in searches
- Tab stops
- Automatic wrapping of long text lines
- Automatic indents for new lines
- Automatic new line insertion when [Enter] is pressed
- Use the cursor position and editor status that was active when the record was last saved or use the system defaults defined at the Editor window
- Preserve or clear existing data at a prompt when you begin typing

Default settings for these features are specified at the Editor Environment window. Use the **Management-Environment-Editor** option on the Main menu to access the Editor Environment window and change the defaults.

You can temporarily override some Editor environment defaults and specify new settings for the current session. To do so, use one of the softkeys listed below:

When you want to:	Press this key:
Change tab stops	[Ctrl-T]
Use the Auto Line Modes popup to:	
Change auto word wrap	
Change auto line indent	[Ctrl-W]
Change auto line insert	

Changing Tab Stops

When you press [Ctrl-T], the current tab stops appear at the top of the screen.

Each bar (|) marks the location of a tab. The cursor rests at the beginning of the line of displayed tabs. To set or remove a tab without disturbing the current position of other defined tabs, the Editor should be in overwrite mode. (If Ed Ins displays on the status line, press [Ins].) With the Editor in overwrite mode:

- To remove a tab, use the arrow keys to move the cursor to the tab you want to remove and press [Spacebar].
- To create a new tab, use the arrow keys to move the cursor to the location you want and press [|] (a shifted backslash).

When you are finished setting the tabs, press [Enter] or [F9] to save the changes and return to your record. To return to the record without saving your changes, press [Esc].

Changing Automatic Environment Features

Depending on the type of record you are editing, automatic environment features can simplify your editing task and minimize your keystrokes. These features operate when editing entry lines in windows and when using the full-screen Editor.

You can press [Ctrl-W] to temporarily change automatic environment features of the Editor. The following popup appears:

Auto Line Modes		
	Mode	On/Off
1>	Insert Mode	On
2	Wrap line	On
3	Insert line	On
4	Indent line	On

Highlight the feature you want to change and press [Enter]. The [Enter] key toggles the feature on and off. When the settings are the way you want them, press [F9] to save the settings and return to the record.

Wrap line determines whether a line of text that is too long to fit in the borders of the window will automatically wrap down to the following line. With word wrap on, you can type continuous lines of text and the Editor will format the lines automatically inside the window. With word wrap off, lines of text too long to fit in the window stay in one line on the screen, with the text tickertaping off to the left as more text is added.

Indent line determines indentation of new lines of text. With auto line indent on, new lines are indented to match the starting point of the previous line. With auto line indent off, new lines begin at the first column in the entry line regardless of the indent of the current line.

Insert line determines whether [Enter] inserts a new line in existing text. With auto line insert on, a new line is inserted and the following lines move down one. With auto line insert off, the cursor moves to the first column of the following line of text. No new line appears.

For example, to write new notes in a text field, you would probably use these settings:

- Auto word wrap ON
- Auto line indent OFF
- Auto line insert ON

To write an outline of a proposal in the full-screen Editor, you might use the same settings except with auto line indent on, to save manually spacing and tabbing to each subheading in your outline.

To write or edit lines of program code in the full-screen Editor, you would probably use these settings:

- Auto word wrap OFF
- Auto line indent ON
- Auto line insert ON

Calling an Operating System File into the Editor

To call and edit an operating system file in the full-screen Editor, use the EDIT command at TCL with the DOS option:

```
EDIT DOS pathname\osfilename
```

For example, to edit an operating system file named MYNOTES.DOC in a directory named DOCUMENTS, enter this command at TCL:

```
EDIT DOS C:\DOCUMENTS\MYNOTES.DOC
```

Advanced Revelation starts the full-screen Editor and displays the file in the Editor window.

When an operating system file first appears in the full-screen Editor, carriage returns and line feeds are displayed as ASCII graphics characters. For ease of editing, you can use the Softkey [Shift-F7] to edit the file in the operating system format. For details, see "Editing Operating System Files" later in this chapter.

Leaving the Full-screen Editor

To leave the Editor, press [Esc]. If you have made changes to the current record since writing the record to the file, Advanced Revelation asks you to confirm that you do not want your changes saved. To discard your changes and leave the Editor, press [Esc] again. To write your changes to the file, press [F9].

Undoing Edits to a Record

To undo edits you have made to a record and start again with the most recent data saved to the file, press [Ctrl-Q]. Before undoing your edits, Advanced Revelation asks you for confirmation. To undo the edits, press [Ctrl-Q] again. To cancel the undo action, press any other key.

Using Defined Blocks of Text

You can define a block of text and copy it once or many times at any cursor position. In addition, by using the Special Modes popup, you can move a defined block to another location in the record or delete it entirely.

Defining a Block

To define a block of text, follow these steps:

1. Move the cursor to the first line of the block and press [Ctrl-B].
2. Use the down arrow key to move the cursor to the block's last line.
3. Press [Ctrl-B] to mark the last line of the defined block. The entire block is defined with a highlight.

What is a Buffer?

Advanced Revelation provides five buffers, numbered Buffer1 through Buffer5, where defined blocks of text can be stored for later use. You can think of a buffer as a clipboard that holds a segment of a field or record. Because you have five buffers to work with, you can hold up to five different blocks at one time.

Buffers are system-wide and stay active between processes. For example, you can store a block of text from one record in a buffer for use later in another record. The buffers are cleared when you exit Advanced Revelation.

Keys for Working With Defined Blocks

Use these keys to work with defined blocks of text:

When you want to:	Press this key:
Mark the beginning and ending of a text block	[Ctrl-B]
Cut the current defined block to a buffer	[Ctrl-F3]
Paste the current block from the buffer to the cursor position	[Ctrl-F4]
Clear the current block definition	[Ctrl-U]
Reformat the current block	[Ctrl-Z]
Use a Special Modes popup to:	
Edit the buffer	[Ctrl-S]
Move the block to cursor position	
Delete the block	

Cutting and Pasting

To cut a defined block to a buffer, press the Cut key [Ctrl-F3]. A message tells you that the block has been cut to the current buffer.

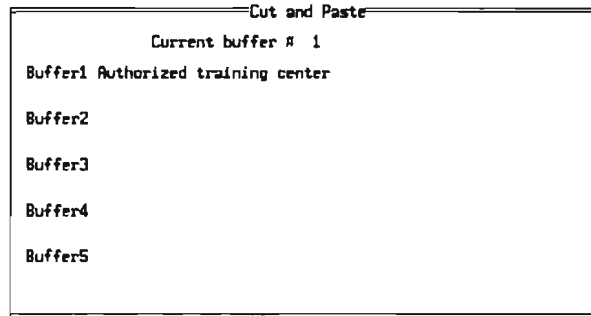
The block that is stored in the buffer can be pasted into any record displayed in the full-screen Editor or at a prompt in a window. To paste the defined block, position the cursor where the stored text should be inserted and press [Ctrl-F4]. The text is pasted where you want it. The text that was cut to the buffer remains available and can be pasted more than once.

Using More Than One Buffer

Use the Special Modes popup to see and edit the buffers. First, press [Ctrl-S] to display the popup:

Special Modes	
1	Move block
2	Delete block
3	Clear text
4	Edit buffer
5	Ascii table

Choose the Edit Buffer option to display the Cut and Paste window.



A flashing cursor rests on the current buffer number. When you cut or paste a block of text, Advanced Revelation always uses the current buffer.

To change the current buffer, enter the new buffer number at the Current buffer # prompt. Press [F9] to save the window.

Editing the Contents of a Buffer

To edit the contents of any buffer, follow these steps:

1. Press [Ctrl-S] to display the Special Modes popup. Choose Edit buffer.
2. Move the flashing cursor to the appropriate buffer and edit it as you would any entry line.
3. Press [F9] to save your changes and return to your record.

Moving a Block

To move a block of text from its current location, follow these steps:

1. Define the block of text you want to move.
2. Place the cursor at the first line of the block's new position.
3. Press [Ctrl-S] to display the Special Modes popup.
4. Choose Move block.

Advanced Revelation moves the block to the location you want.

Deleting a Block

To delete a block of text, follow these steps:

1. Define the block of text you want to delete.
2. Press [Ctrl-S] to display the Special Modes popup. Choose Delete block.

The Editor deletes the block.

Editor Block Reformatting

You can reformat a block of text using [Ctrl-Z] (Reformat) while editing.

Warning! If you press [Ctrl-Z] without marking a block of text first, the entire contents of the window will be reformatted.

To reformat a block of text, mark the block using [Ctrl-B].

Once the block is marked, press [Ctrl-Z]. A message will appear asking you to repeat the keystroke in order to confirm that you wish to reformat. Press [Ctrl-Z] again.

The text in the marked block will be reformatted to fit into the current window width. If you wish the text to be narrower or wider than the current window width, use [Ctrl-F7] (Resize) before reformatting to make the window the appropriate width.

Clearing a Block Definition

When you are finished with a defined block, press [Ctrl-U] to clear the block definition from the screen.

Searching and Replacing

You can search for a specific string of text and find every instance of the use of the string in the record, or you can search for a string of text and replace it with a different string.

By default, Advanced Revelation makes all string searches case sensitive. This means that in order for a character string to match the one you enter, it must appear in exactly the same upper/lower case letter combination. Thus, if you enter "PERSONNEL", the text search will not locate the word "Personnel".

You can change your environment to specify that string searches are case insensitive. When you specify case insensitivity, searches will find all upper/lower case letter combinations of the search string. For example, if you enter "PERSONNEL", the text search will find "Personnel", "personnel", "PERSONNEL", etc.

Case insensitivity is turned on and off at the *Case insensitive search (Y/N)* prompt in the Editor Environment window. Choose **Management-Environment-Editor** from the Advanced Revelation Main menu to access the Editor Environment window. Enter Y (yes) at the *Case insensitive search (Y/N)* prompt to turn on case insensitivity.

Use these keys to search and replace:

When you want to:	Press this key:
Find a string of text	[Ctrl-F]
Repeat the previous Find action	[Ctrl-A]
Find and replace a string of text	[Ctrl-R]

Finding a String of Text

To find a string of text, press [Ctrl-F]. A prompt at the bottom of the screen asks for the text you want to find. Enter the text at the prompt.

Replacing a String of Text

To replace a string of text, press [Ctrl-R]. Prompts at the bottom of the Edit window ask for the text you want to find and for the replacement text. Enter the text you want to find and replace.

- ☛ Your environment determines whether or not the search is case insensitive. However, the replacement text is used exactly as entered. For example, in a case insensitive search and replace operation, a search for "personnel" might find the words "PERSONNEL" and "personnel". If you specify "workers" as the replacement text, the word "PERSONNEL" will be changed to "workers".

Using Value Windows

To display and edit multiple values, sub-values, or text lines, each on its own line, move the cursor to the line that contains the values you want to edit and press [Ctrl-E]. After making your changes, press [F9] and then [Esc] to return back to the record.

When the full-screen Editor displays a record, each field appears on its own line. Multiple values, sub-values, or lines in a text field are separated by these markers:

Table D4-1

Markers Used to Separate Field Values

Type of Value:	Marker:
Multiple values in a multivalue field	ASCII 253
Sub-values	ASCII 252
Text lines	ASCII 251

Editing Multiple Values

Fields that contain multiple values or sub-values can be difficult to read and edit when all the values appear on one line. To display and edit multiple values in a multivalue field each on its own line, move the cursor to the line and press [Ctrl-E]. The Field window appears with the multiple values displayed separately.

You can edit the values as you would any multivalue prompt. To save your changes, press [F9]. To return to the full-screen Editor, press [Esc].

Editing Sub-Values

Sub-values in a multivalue field appear separated by an ASCII character 252. To display and edit sub-values each on its own line, follow these steps:

1. Move the cursor to the line containing the sub-values you want to edit.
2. Press [Ctrl-E] twice. The Value window appears, displaying each sub-value on its own line.
3. Edit the sub-values as you wish.
4. To return to the full-screen Editor, press [F9] to save your changes and [Esc] to return to the previous window. To retain your changes, press [F9] again at the Field window before returning to the full-screen Editor.

Editing Text Lines

Advanced Revelation uses text marks (ASCII 251) as delimiters to maintain the format of formatted text fields. When a text field is displayed on the full-screen Editor, the markers Advanced Revelation uses to format the text are displayed.

To display the field formatted in the same way it would be displayed in a window, press [Ctrl-E] three times. Edit the field as you wish. To save your changes, press [F9]. To retain your changes and return to the full-screen Editor, you must press [F9] and [Esc] at each window until the full-screen Editor appears.

Subdividing a Field

Each time you press [Ctrl-E], Advanced Revelation further subdivides the current value and displays the subdivision in a new window. Edit the value in the window as you wish. To retain your changes, you must press [F9] at each window as you return through the Value windows to the full-screen Editor.

Using Softkeys in the Full-screen Editor

The following table shows the Softkeys defined for the full-screen Editor. You can execute softkeys directly by pressing the appropriate keys or indirectly by choosing options from the Softkey popup. To use the Softkey popup, press [F6].

When you want to:	Use this Softkey:
Edit a different record	[Shift-F1]
Edit an \$INSERT record at the current cursor position	[Shift-F2]
Check record size	[Shift-F3]
Shift a block's indentation	[Shift-F4]
Convert defined block to upper case	[Shift-F5]
Convert defined block to lower case	[Shift-F6]
Edit a record in DOS format	[Shift-F7]
Convert a record to DOS or Advanced Revelation format	[Shift-F8]
Save and compile a record	[Shift-F9]

Displaying Additional Records to Edit

To display and edit records other than those in your current browse list, use one of these methods:

- **To display any record**, press the Softkey [Shift-F1]. A window appears in which you enter the name of the file and the record you want to edit.
- **To display additional records** contained in your current file, press [Alt-I] and add the record keys to your browse list.

For example, suppose you have a file named SOURCE_BP that contains R/BASIC programs. While writing a new program, you realize that with a little revising you can use a block from another program as a block in your current file.

First, you press [Shift-F1] and enter SOURCE_BP as the file to edit. Then you enter the name(s) of the record or records that contain the blocks you want. Press [F9] and Advanced Revelation displays the first record in the Editor window.

You define in a block the lines you want to use in your original file and cut them to the buffer. To return to your original records, press [Esc].

Editing \$INSERT Records

A \$INSERT statement directs the compiler to insert R/BASIC source code from another record into the current record. For more information on using \$INSERT statements, see the "Programming with R/BASIC" chapter in the *R/BASIC* section.

To edit the record named in an \$INSERT statement, move the cursor to the line containing the \$INSERT statement and press [Shift-F2]. Advanced Revelation displays the record so that you can edit it. When you are finished editing the record, press [F9] to save your changes and [Esc] to return to your original record.

Checking the Record Size

The maximum size of source code that can be compiled is approximately 34,000 bytes. To check the size of your current record, press [Shift-F3]. Advanced Revelation displays the current record size.

Shifting a Block's Indentation

To shift a group of lines so that they start at a new column position, follow these steps:

1. Define the block of lines.
2. Move the cursor to the column where you want the block to start.
3. Press [Shift-F4].

The block shifts to the new indentation.

Converting a Block to Upper or Lower Case

In some situations, you may want to convert a block of text to all upper case or all lower case. For example, when writing program code, you can write all your code and comments in upper case, then convert the comments to lower case so they are easier to find and read.

Or, because the Find action is case sensitive, you may want to convert a block of text to all one case when you are searching for a specific string of characters in the block. For more information on the Find action, see "Searching and Replacing" earlier in this chapter.

To convert a block of text to all upper case, first define the block and then press the Softkey [Shift-F5].

To convert a block to all lower case, define the block and then press the Softkey [Shift-F6].

Editing Operating System Files

When you create and edit Revelation records, Advanced Revelation places special markers between fields, values, and sub-values. In the full-screen Editor, each field appears on a single line. When you save the record, the lines are written to the record as individual fields.

In contrast, operating system files have carriage returns and line feeds between the lines of text. By default, Advanced Revelation displays carriage returns and line feeds in the full-screen Editor as ASCII graphics characters.

To edit an operating system file, use the softkey [Shift-F7]. Advanced Revelation displays the file so you can edit it.

Converting to Operating System or Advanced Revelation Format

When Advanced Revelation converts an operating system file to Advanced Revelation format, the carriage returns and line feeds in the file are replaced by Advanced Revelation field markers. In the same way, when Advanced Revelation converts an Advanced Revelation formatted file to operating system format, the field markers are replaced by carriage returns and line feeds.

- Use DOS format to edit material that will be written to an operating system file and not to an Advanced Revelation or bonded file. For example, your AUTOEXEC.BAT file should be edited in DOS format.
- Use Advanced Revelation format to edit material that will be written to an Advanced Revelation or bonded file.

To convert a file's format, press the Softkey [Shift-F8]. Use this key as a toggle to switch from operating system format to Advanced Revelation format and back again.

Saving and Compiling Programs

To save a record, press [F9]. To save and compile a program source record, press the Softkey [Shift-F9].

When you press [Shift-F9], Advanced Revelation saves the current record and immediately starts the R/BASIC compiler to compile the record. For more information on the compiler, see the "Programming with R/BASIC" chapter in the *R/BASIC* section.

If the compile is unsuccessful, Advanced Revelation returns you to the Editor with the cursor on the first line that contains an error. The Status Lines display the error message.

Using the Keystroke Translator

The Keystroke Translator displays your typed keystrokes in the format Advanced Revelation uses to create a set of captured keystrokes. For example, when you press [↑] with the translator on, instead of moving the cursor up one line Advanced Revelation displays the following:

```
{UP}
```

To toggle the translator on and off, press [Ctrl - —].

Displaying the ASCII Character Codes

To display a table of the ASCII character codes, follow these steps:

1. Press [Ctrl-S] to display the Special Modes popup.
2. Choose Option 5 ASCII table.

Advanced Revelation displays the table. To return to your record, press [Esc].



Chapter E 1

Advanced Revelation Files and Records

Whether you are new to database management systems or just new to Advanced Revelation, there are terms and concepts in this chapter that you will want to be familiar with. Because many of the basics of Advanced Revelation database management are summarized here, you will probably find yourself returning to this chapter time and again to solidify your understanding of these concepts.

About Files & Records	E 1 . 1
Terminology: Files, Records, and Fields	E 1 . 1
Characteristics of Advanced Revelation Files	E 1 . 2
Characteristics of Advanced Revelation Records	E 1 . 5
Characteristics of Advanced Revelation Fields	E 1 . 6



About Files & Records

Advanced Revelation and other programs of its type were designed to help users give order to collections of data, or databases. As a group, these programs are known as database management systems (DBMS) or database managers. The database that you are organizing can be as simple as the contents of your address book or as complex as the parts list for an airplane.

This chapter is a general overview of the basic concepts of database management, Advanced Revelation style. You will find information here about how Advanced Revelation stores information, and about how an Advanced Revelation system is organized.

If you are new to database management, you will want to learn some of the basic principles and terminology used in Advanced Revelation. Even if you have experience with another database manager, you will find it useful to review the Advanced Revelation approach to data management.

The best way to use this chapter is to read through it casually the first time through. Familiarize yourself with some of the basic concepts, and get an idea of what capabilities are available to you in Advanced Revelation.

Later, when you need information about a specific topic such as files or data typing, return to this chapter and read about that topic in detail. You will find that other sections of the documentation will refer you back to this chapter whenever you need background information about a topic.

- ☛ If you are using an Environmental Bond, some features of Advanced Revelation files may work slightly differently than described here. For information on Environmental Bonding in general and on the characteristics of specific bonds, see the section *Environmental Bonding*.

Terminology: Files, Records, and Fields

The information that you store and manage for your Advanced Revelation applications is stored in files. For example, your application might contain files of your customers, of parts used in assembling components that you manufacture, or of books available in a library. Your application can contain any number of files. A complex application such as a financial, personnel, or manufacturing system might contain hundreds of files.

Individual entries in a file are called records. In a file containing customer information, an individual customer will be a single record. In a file containing information on the books in a library, each book will have its own record in the file.

Records are in turn divided into fields. A field contains a single component of the record. For example, a customer record might be divided into fields for the customer's name, address, phone number, and so forth. A conceptual picture of how a file is arranged looks like this:

	Phone	Date added	Dues paid
Record 1	(206) 643-9898	05/25/86	\$1,000
Record 2	(206) 443-4747	01/08/88	\$2,500
Record 3	(206) 628-0888	09/17/87	\$1,750

If you have worked with other programs, you may know some of Advanced Revelation's terminology by other names:

Advanced Revelation Name	Synonyms
Files	Tables
Records	Rows
Fields	Columns, attributes

Characteristics of Advanced Revelation Files

An Advanced Revelation application can contain any number of files. Files that store data are generally referred to as data files. However, the distinction between data files and other types of files is one of convenience. There is no fundamental difference between a file that stores data, a file that contains programs, and a file that contains the characteristics of a data entry window. Each record in a CUSTOMER data file contains information on a single customer, each record in a programs file contains a single program, and each record in the TEMPLATES file defines a single window. In Advanced Revelation, the purpose of a file is defined by how it is used, not by the file type itself.

File Types

Advanced Revelation lets you use different file types. Two file types are available in Advanced Revelation: Linear Hash files (default) and ROS files. Linear Hash is the standard filing system used by Advanced Revelation and is recommended for almost all applications.

If your Advanced Revelation applications are tied to another environment (for example, dBASE or ASCII) through Environmental Bonding, you can also create and use files in the file type for that environment. For information, see the section *Environmental Bonding*.

In addition, you can get direct access to operating system (DOS or OS/2) files.

Linear Hash Files Linear Hash files are self-sizing, meaning they expand and contract automatically as data is added and deleted. In addition, they are self-optimizing, so access to data is always as efficient as possible. Linear Hash files represent the best

balance between speedy access to data and minimal cost in disk space.

Linear Hash files were specifically designed to be used on networks. If you are using Advanced Revelation, you must use Linear Hash files to share data on a network. The additional advantages mentioned above — self-sizing and self-optimizing structure — are particularly beneficial on a network, because disk access can be a major source of slowdown. Linear Hash files also offer an option to enable you to match certain characteristics (packet or cluster size) of the network you use.

When used in a single-user environment Linear Hash files have an added benefit: the Linear Hash system stores some information in memory to speed up access to records. This helps reduce the number of disk reads that might otherwise be required to access data.

ROS Files The second filing system offered by Advanced Revelation is the ROS system. ROS is a file type used primarily for small files (less than 100K bytes in size) that are not shared in a network.

ROS files can be very fast because the data stays resident in memory (is “buffered” or “cached”) rather than requiring disk access for data retrieval. Advanced Revelation will store as much of the ROS file in memory as it can. As a result, small ROS files are especially efficient, because the entire contents of the file will fit into memory.

ROS files can be used on a network, but they have special requirements not applicable to Linear Hash files. First, ROS files *do not support record or file locking*, which means that they cannot be shared without risking data corruption. They can still be useful, however, for small files that are only used for lookup or for program files from which data is read but to which it is never written.

Second, ROS files will read up to 64K bytes at a time, but networks typically only transfer somewhere between 1K and 4K at a time, so ROS files might not be advantageous on a network.

Files Outside of Advanced Revelation

In addition to these file types, Advanced Revelation allows you to access data in the formats of other products, such as dBASE III or a database server. This capability — Environmental Bonding — means that there is really no constriction on how you choose to store data. For more information on Environmental Bonding, see the section *Environmental Bonding*.

The Data Dictionary

Each Advanced Revelation file has associated with it a data dictionary, or dictionary. Each field that you define has an entry in the dictionary. The dictionary for a file is itself a file and is the central source of information about the file's field attributes, indexing, and other information.

Volumes

You can create and maintain Advanced Revelation files on virtually any storage device: your local hard drive, remote drives, floppy disks — almost anywhere. If you can assign a drive name to a device, you can store and access data there.

To create and access files on your system, you associate them with a “volume”. The volume specification consists of two pieces of information about a file: the file's location and its file type.

For more information on volumes, see the chapter “Advanced Revelation Volumes” later in this section.

Accounts

An account is a collection of files that contain related information, such as payroll or inventory records. Accounts are used to group information together so it can be easily located and protected.

You can use accounts to protect sensitive information and to segregate different applications, which is particularly important in a networking environment. Each account has its own set of security restrictions, which protects access to the account.

For more information on accounts, see “Users and Accounts: Planning Access and Security” in *Developing an Application*.

File Synonyms (Qfiles)

In Advanced Revelation, you can establish a synonym (Qfile) for an existing file and use the synonym file as you would the original. Generally, when you are working in one account, files in other accounts are not available to you. Qfiles enable you gain access to files in those other accounts.

For information on creating file synonyms, see “Managing Files and Records” in this section.

Editing Operating System Files

In addition to the other file types accessible through Advanced Revelation, you can use Advanced Revelation to access data in operating system files. You can, for example, read and edit your DOS AUTOEXEC.BAT file. When you do so, DOS is

treated as a single file, and DOS files are treated as records. Operating system files are limited to 64K except when using R/BASIC.

For more information on editing operating system files, see “Using the Full-Screen Editor” in *Getting Started* and “Programming with R/BASIC” in *R/BASIC*.

File Names

The conventions for naming Advanced Revelation files and fields are few:

- They may be any length up to 50 characters.
- They may contain upper-case alpha and numeric characters.
- They may use “.” (period) and “_” (underscore) as delimiters (for example, CUSTOMER.HISTORY).

If you want your file and field names to be SQL compatible, it is a good idea to limit field length to 18 characters and to use only “_” (underscore) as a delimiter.

Advanced Revelation vs. Operating System Names

When you create a new file, that file has two names: the name you give it in Advanced Revelation and its operating system name. The rules for naming files in your operating system are considerably more restrictive than the rules for naming Advanced Revelation files. DOS will not let you name a file CUSTOMER.HISTORY, for example.

To find the DOS file name of an Advanced Revelation file, you can check the volume directory. This is itself a file containing information on the relationship between Advanced Revelation file names (for example, CUSTOMER.HISTORY) and operating system file names (for example, REV10000.LK).

For information on listing volume directories, see “Advanced Revelation Volumes” later in this section.

Characteristics of Advanced Revelation Records

Records are the building blocks of files. There is no limit on the number of records that you can keep in a file, assuming your computer has sufficient storage space. You can store up to 64K bytes (65,535 characters) in a record, which is about 25 pages of typed, single-spaced text.

Record Keys

Each record in a file has a field that is used to uniquely identify that record within the file. The record key, also known as the primary key, is the means by which any record can be distinguished from any other record. Every record must have a key, and no two records in the file can have the same record key.

For example, you might designate the CUSTOMER.NUMBER field to be the key field for your customer records. Each customer will have a unique entry in the CUSTOMER.NUMBER field, so that you can tell one customer from another.

Indexing

Unless you have a tiny file, you will need a means of retrieving records other than simply knowing every key for every record in the file. The alternative to memorizing record keys is to index fields whose values you do know. In the CUSTOMER file, the record keys might be six-digit numbers that you have not committed to memory. However, you do know the names of your customers, so you index the name field. You can then retrieve records for those customers by entering their names instead of their record keys.

There are three benefits to indexing. One is the ability to retrieve data even if you do not know the record key. The other two are speedier processing time when sorting and when selecting records for reports. Be wary of over-indexing, though, because indexes take up storage space and processing time.

For more information on indexing, see “Creating and Maintaining Indexes” in *Developing an Application* and “Using Indexes” in *Accessing Data*.

Characteristics of Advanced Revelation Fields

If records are the building blocks of files, then fields are the building blocks of records. This topic discusses the essential attributes of fields, including some that are unique to Advanced Revelation.

Variable-length Fields: The Size and Number of Fields in a Record

In Advanced Revelation, there is but one restriction that controls both the maximum number of fields in a record and the maximum size of an individual field: a record can be no longer than 64K bytes (65,535 characters). Each empty field in a record takes up one byte, so you can have one field containing 65,535 characters, or 65,535 fields containing no data, or anything in between.

Each field in a record is *variable* in length — there is no fixed length established for a field. Although you will specify a display length and justification when defining the field, there is no limit on how many characters you can enter into a field. The only limit on field length is that the entire record cannot exceed 64K.

Advanced Revelations’s variable-length fields have two major advantages over the fixed-length fields of most database managers. First, you need not specify a fixed number of characters. In many systems, you must guess in advance how long the longest piece of data will be for each field.

Second, you can change the structure of a variable-length record very easily. Unlike most database managers, Advanced Revelation allows you to add or delete fields or change their characteristics at any time, even after data has been entered in the file. For example, you might decide to add a "comments" field to a customer record. You can do this even if you have thousands of records in the customer file already.

Moreover, such alterations do not require any special processing before the file can be used again. Many database management programs require that you restructure a file after adding, deleting, or modifying fields and will not let you use the file until you have done so.

Multivalued Fields

You can store *multiple values* or *repeating data* in a single field. For instance, if there is more than one phone number for a customer or if a part consists of multiple subassemblies, you can store all entries in a single field. Again, the only limit is that the entire record cannot be longer than 64K.

Conceptually, a multivalued field looks like this:

Name	Address	City	State
Larry Smith Louise Fay Paula Baker	12 Roy St	St. Oberlin	NJ

↑
Multi-valued field

Two or more multivalued fields with related data can be tied together, or "associated". Imagine a file that contains data on client organizations. In each record of that file, there is one multivalued field containing the names of contacts and another containing their direct-line phone numbers. In Advanced Revelation, you can make these two multivalued fields associated, so that the first contact is tied to her phone number, the second to his phone number, and so on.

Group Fields

For the sake of convenience, you can define a field as a group field and specify two or more other fields as elements in the group. This group field can then be used when you run a report, saving you the trouble of naming, one by one, each field contained in the group field.

For example, in your STAFF.PAYROLL file you can define a single field, PAYROLL.SUMMARY, that includes every field that you need for a payroll report (EMPLOYEE.NAME, SALARY, FEDERAL.TAX, STATE.TAX, FICA, etc.). When you run the report, you can simply specify the group field PAYROLL.SUMMARY and produce a complete report without naming the fields individually.

Synonym Fields

You can create a field that is a synonym of another field. You then have two definitions for the same data. There are two primary advantages to synonym fields. They can be used to substitute a short field name for a long one, and they can be used to report data in a slightly different format (truncated from 40 to 20 characters, for example) without affecting the contents of the original field. You can define any number of synonyms for a field.

Symbolic Fields

Symbolic fields, also called calculated fields, are not associated directly with data in the record. Instead, they calculate a value based on a “formula”. This formula can be used to calculate totals (an INVOICE.TOTAL field, for example), look up values (to extract data from a related file), or perform other processes. Using the payroll example again, you could define the fields for payroll taxes as symbolic fields that calculate taxes based on salary. These calculations could include comparing salary in the STAFF.PAYROLL file with tax tables in other files.

The formula for a symbolic field can be up to 34K, and a record can contain any number of symbolic fields.

Field attributes

The definition of each field consists of up to fifty attributes. These include display length (as distinct from actual length, which is unlimited), justification (based on display length), single or multivalued, and validation patterns.

- | | |
|-----------------------|--|
| Field name | Each name that you define in a file has a unique name (see “Field Names” above). It only has to be unique within the file — you can use the same name in another file if you need to. |
| Display length | <i>Display length</i> refers to the number of characters that are output when running reports. It is <i>not</i> a limit on the size of the field. The only field size restriction is the maximum record size, 64K. |
| Justification | <i>Justification</i> , indicating whether data is to be left or right justified or centered, refers to the orientation of data within the display field and imposes no limits on the size of the field. |

Output format You may choose to store certain types of data in a compact format. This is most common with dates and times, money, phone numbers, and the like. The *output format* is used to specify how this data will look when it is displayed.

For example, you can enter a date as numbers (100968) and convert it to any of a variety of date formats for output: October 9, 1968, 9 Oct 1968, 10/09/68, 10-09-68, 09-10-68, etc.

Validation patterns To ensure that data is entered properly, you can define *validation patterns*. As data is entered, it can be checked against a validation pattern and rejected if it does not match. For example, you could define a validation pattern that checks data entered into a phone number field. If you do not enter ten numeric characters, you will see an error message and will have to re-enter the phone number.

Data Types

When you define a field, you will indicate the field's "data type". The data type — such as DATE, TIME, or DOLLARS — tells Advanced Revelation about how you want to store and display the information in a field.

Advanced Revelation has predefined display length, justification, output format, and validation patterns (described above) for eleven commonly-used data types. The default values are advisory rather than mandatory, so you can change them as desired.

Table E1-1
Advanced Revelation Data Types

Type	Meaning	Example
BOOLEAN	True/false	1/0, Y/N
CHAR	Fixed # of alphanumeric chars	NY, WA
DATE	Date	01 JAN 1989
DATETIME	Date + time	01/01/89 10:00:00
DECIMAL	Number with fixed decimal place	1.1, .50
DOLLARS	Money format (US dollars)	\$100.00
FLOAT	Number with variable decimal places	3.14159...
INTEGER	Whole number	98101, -3
TIME	Time	10:00AM
VARCHAR	Varying number of characters	ABCDE...Z
TEXT	Characters to any length	In the Beginning...

In addition to these basic data types, an Environmental Bond may offer you others that are used in foreign environments. For more information, see the section *Environmental Bonding*.

**Extra Information for
Data Type**

If you specify some of these data types, you are required to provide additional information. For example, if you choose the data type CHAR, you will be asked for the length. On the other hand, a data type such as DATE does not require any additional information, although you can alter the output format if you choose to.

Chapter E 2

Managing Files and Records

In this chapter you will learn how to manage Advanced Revelation files. Here you will find information on defining and creating files and fields. In addition, there is information on Advanced Revelation file attributes, creating file synonyms, modifying Advanced Revelations vocabulary, transaction processing, and on the system files used in Advanced Revelation.

Defining Files and Fields	E 2 . 1
Defining a File Using the Definefile Window	E 2 . 1
Advanced Revelation File Attributes	E 2 . 13
Creating File Synonyms	E 2 . 15
Adding or Modifying Advanced Revelation Commands (the VOC File) . . .	E 2 . 16
Transaction Processing	E 2 . 18
Advanced Revelation System Files	E 2 . 28



Defining Files and Fields

Before you store data in a file, you must first “define” the file. There are four ways to define a new file. These are:

- The Definefile window. Details about using this window appear below.
- The DEFINEFILE command at The Command Level (TCL). If you use this command, you can enter all the information about the new file in one line at the TCL prompt. For details on how to use the TCL DEFINEFILE command, see the chapter “TCL Command Reference” in the *TCL* section. Before you use the command, however, you should read the following discussion of the Definefile window for general information on how to define a file.
- Paint. This Advanced Revelation tool allows you to define a new file at the same time as you are painting a new window. Paint is discussed in detail in the *Developing an Application* section.
- The SQL command CREATE TABLE. For details on using this command, see the *SQL* section.

Defining a File Using the Definefile Window

The Definefile window allows you to create a new file and to define new fields by filling in information at prompts. You can also use the Definefile window to change the definition of a file.

- Some Environmental Bonds restrict the changes you can make to the definition of an existing file. For more information, see the topic “Making Changes in Foreign Files” following, and see your bond documentation.

The Definefile window is arranged so that you define the file using a set of discrete steps:

- Creating the dictionary of the file. In this step you indicate the Volume for the file (thereby telling Advanced Revelation the file type and location of the file), and define any special attributes for the file.
- Establishing field definitions for the new file. This includes naming the new fields, as well as entering information such as data type, length, and justification.
- Creating the data portion of the file. When you save your file definition, you can create the data file itself.

In addition to these basic functions, the Definefile window provides you with additional capabilities. These are:

- Defining symbolic and group fields, multi-part keys, and field synonyms.
- Displaying the Dictionary window in order to specify detailed information about each field in your file.
- Displaying windows that collect additional information used to define files if you are using an Environmental Bond.

As you read the following discussion about Definefile, refer to this illustration:

Define file

File name
Volume
Bond name
Foreign file name

Created
State
Allow File Delete (Y/N)?
File Attributes

Field name	T	Pos	Data type	Column Heading	Len	J	S/M

Information-only Prompts

There are prompts in the window that are there just to give you information about the file. These prompts are for information only — you cannot change the entries. Information-only prompts tell you this information about the file:

- *Bond name*: the name of the file type (bond) being used to define the file.
- *Foreign file name*: the name by which the file is identified in the foreign environment.
- *Created*: The time and date that the dictionary was created.
- *State*: Whether the data file has been created, or whether you are still in the process of defining it.

Defining the Dictionary

When you create the dictionary of the file, you indicate such information as the name of your file, the file type, and optional information about special file attributes.

- The instructions that follow are oriented toward defining Advanced Revelation files. However, if you are using an Environmental Bond, there might be slight variances in the file definition process. For more information, see the topic “Using Definefile with Environmental Bonds” later in this chapter. Any differences in file definition are also discussed in the bond documentation that accompanies your bond.

To create the dictionary, follow these steps:

1. Display the Definefile window by choosing the options **Tools-Files-Definefile** from the Advanced Revelation Main menu.
2. Enter a name for the file at the *File name* prompt. Alternately, you can skip the *File name* prompt and move directly to the *Volume* prompt.

- ☛ For information on file name conventions, see the chapter “Files and Records”.
- 3. At the *Volume* prompt, enter the name of the volume for the file you are defining. The default is DATAVOL. If you wish to see a list of possible volumes, press [Shift-F2].
 - ☛ Remember that a volume identifies both the location and file type of a file. For more information on volumes, see the chapter “Advanced Revelation Volumes” in this section.

Notice that as soon as you enter a volume, the prompt *Bond name* is filled in. The information at this prompt is derived from the name of the volume you just specified. *Bond name* is an information-only prompt.

- 4. If you did not enter a file name in step 1, move to the *File name* prompt and enter it now. If you already entered a file name, skip this step.

If you press [F2] at the *File name* prompt after you have already entered a volume, you can see a list of all the files that have already been created on that volume.

- 5. If you wish to specify attributes about the particular file type you are creating, enter these at the *File Attributes* prompt.

Different file types allow you to specify different attributes (some file types do not allow file attributes). Press [F2] to see a window of possible file attributes for the current file type. If you are unsure about how to respond to the prompts, press [F1] for Help, and [F2] for options.

- ☛ This prompt is optional. If you leave this prompt blank, the file will be defined using default attributes. For more information on these attributes, see the documentation for the file type you are creating. Information about attributes for Advanced Revelation files appears later in this chapter.

When you have filled in all the prompts in the file attribute window, press [F9] (Save) to return to the Definefile window. The attributes that you specified will appear at the *File Attributes* prompt in the Definefile window.

- 6. When you are finished with the file attributes, press [Enter]. The Definefile window will then create the dictionary of your new file. You will see a message similar to this:

Creating DICT.filename

Notice that two more prompts are filled in automatically. Both of these are information-only prompts. The *Created* prompt displays the time and date when the dictionary was created.

The *State* prompt displays the status of the file. Because you are creating a new file, the prompt displays the word “Defining”. If you were using the Definefile window to view or alter the definition of an existing file, the prompt would display the word “Defined”.

Defining Fields

Once you have created the dictionary for the file, you are ready to define the individual fields for your file. To do so, follow these steps for each field you are defining:

1. At the *Field name* prompt, enter a name for the field. Be sure not to include a space in the field name.

☛ For information on field name conventions, see the chapter “Advanced Revelation Files and Records”.

2. Indicate whether this field is a primary key for your file. If you have not yet identified a key, the window will ask you “Is this the key field?” If so, answer “Y”. If this is not a key field, answer “N”.

☛ This prompt will display each time you define a field until you have identified the primary key for your file.

Once you have filled in a field name, the next two prompts are filled in automatically. The first prompt, *T*, identifies the Dictionary type of your field (real field, symbolic, or group). The default value here is “F”, which identifies a real field (one that contains data).

The second prompt, *Pos*, identifies the position of the field within the record. For example, if this is a key field, the position is zero. If this is the first field in the record, *Pos* will contain the value 1, and if this is the second field, *Pos* will contain the value 2.

☛ Information about creating multi-part keys appears later in this chapter.

3. At the *Data type* prompt, enter the name of the data type for your new field. For example, if you are creating a numeric field with no decimal places (such as a zip code), enter INTEGER. Press [F2] to see a popup of available data types.

Some data types require both a name as well as additional information about the type. For example, the data type CHAR requires that you enter a length for the field.

If the data type you have chosen requires additional information, you can enter it using parentheses after the data type. For example, to indicate a CHAR field of nine characters, enter CHAR(9).

If the data type requires additional information and you do not enter it, you will see another window. Fill in the information as prompted, and press [F9] to complete the data type prompt.

4. Once you have indicated a complete data type, the Definefile window fills in the remaining prompts:
 - *Column Heading* is the name that will appear at the top of a column when you use this field in a report.
 - *Len* (length) is the display length for this field in a report.
 - *J* (justification) tells you how the data will be justified in a report: left, right, centered, or text (paragraphs).
 - *S/M* (single/multivalued) allows you to specify "S" for a single-valued field, and "M" if you wish to allow multiple entries (repeating data) in a field.

The values at these prompts are derived from the data type you just entered.

Changing Field Definitions

As soon as you enter a data type for a field, the Definefile window fills in the rest of the prompts for that field. Each data type has different default values for the remaining fields. For example, a field with the data type DATE fills the prompts in using these values:

Prompt	Value
<i>Column Heading</i>	(field name)
<i>J</i>	R
<i>Len</i>	11
<i>S/M</i>	S (single)

To change any of these values, follow these steps:

1. Make sure you are on the *Field name* prompt of the field you wish to change.
2. Press [Enter] or [→] to move across the window to the prompt you wish to change.
3. If you are not already in Edit mode, press [F4] (Edit) to enter Edit mode (the cursor should be small), and then fill in the new value for that prompt. Press [Enter] or [→] to move to the next prompt for that field.

Saving your Definition

When you are through defining fields for the file, press [F9] (Save). The Definefile window will ask you:

Are you finished defining the dictionary?

Answer “Y” or “N”. If you answer “Y”, you are telling the Definefile window that it can create the data portion of your file. By answering “N”, you tell the window that you will continue defining fields another time.

- ☛ Most bonds will not allow you to alter a file — by either deleting or adding fields — once you finish the definition. For bonded files, make sure you have a complete definition before answering “Y” to this prompt.

Abandoning your Definition

As you work with the Definefile window, changes you make in the window are reflected directly in the dictionary of the file. If you wish to abandon changes, exit the Definefile window using [Esc]. Any changes you have made are erased, and the dictionary remains the way it was before you entered the Definefile window.

Using Definefile with Environmental Bonds

If you are using an Environmental Bond you can use the Definefile window to define files in the foreign environment. For example, using the dBASE bond, you can create a new dBASE file.

Defining files for a bonded environment is almost exactly like defining Advanced Revelation files. There are just a few differences. These are:

- The foreign environment may have restrictions on the names of files and fields.
- The foreign environment will offer you additional data types.
- Finally, the foreign environment may restrict the changes you can make to an existing file.

Foreign Files and Field Names

When you define a file using the Definefile window, you can name the file and the fields almost anything you like. A foreign environment, however, may not be as flexible about file and field names as Advanced Revelation.

Most of the time, the name that you assign in Advanced Revelation will be sufficient for both Advanced Revelation and the foreign environment. Sometimes, however, you may assign a file or field name that is too long or is otherwise not suitable for that environment.

In that case, the Definefile window allows you to specify *two* names for a file and for fields. The first name is the one you assign in Advanced Revelation. The second name is the one that the file or field will be known by in the foreign environment.

Specifying a Foreign Name

If you are defining a bonded file and use a name that isn't permitted in the foreign environment, Advanced Revelation will create a foreign file or field name for you. You will see a message similar to this:

```
name is not a valid name in the bondname
environment.
    It has been mapped to:
<new_name>
```

The name you see in the *new_name* position will be the name by which the field will be known in the foreign environment. (The Advanced Revelation field name remains the same.) If you approve of the suggested name, press [F9] to accept it. If you do not like the name, enter a new one and press [F9] to save it. The bond will make sure that the new name you enter is valid.

Foreign Data Types

In addition to the basic data types available in Advanced Revelation, an Environmental Bond will offer you bond-specific data types. For example, a bond might offer you a very high-precision number, or a special variant on the DATE type.

Bond-specific data types allow you to choose a foreign data type directly. For example, if you are using the dBASE bond, rather than choosing between the Advanced Revelation data types INTEGER and FLOAT, you can choose the dBASE data type DBASE_N (numeric).

If the bond offers additional data types, you can fill these in at the *Data type* prompt. To display and choose a foreign data type, press [F2] at this prompt. The popup that appears will display both the Advanced Revelation types as well as the data types available in the bond you are working with. Highlight the one you want, and press [F9] to return your choice to the Definefile window.

Making Changes in Foreign Files

Most foreign environments will not allow you to add or delete fields after the file has been created. If you attempt to make such a change to an existing file, you will see a message similar to the following:

```
Error while writing or deleting
Dictionary record fieldname.
File structure cannot be modified
while the data file contains data.
```

The change you attempted will not be made to the file.

As a rule, changes to the structure of the file must be made in the foreign environment itself. See your bond documentation for more information.

Additional Functions in the Definefile Window

Although the Definefile window is simple, it provides you access to almost any facility you require in order to define a file. The following topics address the remaining capabilities of the Definefile window.

Defining a Symbolic Field

When you define a field in the Definefile window, by default you will create an “F” type field (a field that has data associated with it). However, you can also use the Definefile window to define symbolic (“S”-type) fields.

Instead of pointing to a specific position in the record, a symbolic field uses a “formula” as its definition. For example, you might define a symbolic field called INVOICE_TOTAL, one that calculates the total value of an invoice.

To define a symbolic field using the Definefile window, follow these steps:

1. At the *Field name* prompt, enter the name of the symbolic field. The same naming conventions apply to symbolic fields as to “F”-type fields.

2. As soon as you enter the name, the cursor will skip to the *Data type* prompt. Press [Shift-F6] to tell the Definefile window that you want to create a symbolic field.
3. You will see a message that prompts you to confirm that you want to convert the field from a "F" to an "S"-type. Press [Enter] to confirm.
4. A Formula window appears. Enter your formula at the window. If you need help in entering your formula, press [F2] to display the formula builder. When you are done, press [F9] to save your formula and return to the Definefile window.

☞ Note that an S (symbolic) now displays at the *T* prompt. Also, the number under *Pos* disappears. Because a symbolic field uses a formula as its definition, it is not associated with a particular position in the record.

For more information about using formulas, see the chapter "Building Formulas" in the section *Developing an Application*.

5. At the *Data type* prompt, enter a data type appropriate to the data that your formula will calculate. For example, if your symbolic field will calculate a dollar amount, choose the data type DOLLARS.
6. The rest of the prompts for your symbolic field will be filled in automatically based on the data type. You can edit these in the same way you would for an "F"-type field.

☞ See "Changing Field Definitions" earlier in the chapter for information about editing the values at these prompts.

Editing the Formula

You can view or edit the formula for a symbolic field at any time. To do so, place the cursor on the symbolic field you wish to edit, and press [Shift-F6].

Defining Multi-part Keys

When you define a file, you must always specify a field that is the key to the file. A multi-part key is defined when a *combination* of fields in the file is used as the key.

To create a multi-part key, follow these steps:

1. Create the first part of the key by entering in the name of the field at the *Field name* prompt.

2. The cursor will move to the *Data type* prompt. Exit Edit mode by pressing [F4] (Edit) (you should see a full-sized cursor).
3. Press [←] to move back to the *Pos* prompt. Enter a key position number at the prompt, like this:

0*1

In this case, “0” means that the field you are creating is a part of the key. The “1” indicates that it is the first part of the key. Press [Enter] when you have entered the key position information.

4. At the *Data type* prompt, enter the data type of your key.
5. The rest of the prompts will be filled in automatically based on the data type. You can change the values at these prompts by editing them as you would for any field.

☛ See “Changing Field Definitions” earlier in the chapter for information about changing these prompts.

6. Enter in the second (and subsequent) parts of the key by repeating steps 1 through 5. However, at the *Pos* prompt, enter in the appropriate key position information. For example, the two key position values:

0*2

0*3

represent the second and third parts of a multi-part key, respectively.

Defining a Synonym for a Field

After you have defined an “F”-type field, you can define as many synonyms for that field as you like. This allows you to treat the same data in different ways for reports and windows.

For example, perhaps the first definition you create is for a field called COMPANY_NAME, with a length of 35 characters, and a justification of “L”. However, if you frequently display the field in reports to the screen, you might define a synonym called CO_NAME that is only 20 characters wide, but has a justification of “T”. Both fields define the same data, but the field CO_NAME displays it in a narrower column on a report.

To define a synonym field, follow these steps:

1. Place the cursor on the field for which you wish to define a synonym (the "master field").
2. Press [Shift-F8]. A new row will open immediately following the field for which you are defining a synonym.
3. You will see a message that prompts you for the name of the synonym field. Enter in a name, following the usual field name conventions.
4. The Definefile window will fill in the value " (a double quote mark) at the *T* prompt to indicate that this is a synonym. It will also copy the position of the master field to the *Pos* prompt for the synonym field.
5. The cursor will move to the *Data type* prompt. Enter a data type that is appropriate to the synonym field you are creating.

☞ The data type of a synonym does not have to match that of the master field.
6. The rest of the prompts for that field will be filled in automatically based on the data type. You can edit these for a synonym as you would for any other field.

The position (the *Pos* prompt) of a synonym always matches that of the master field. The master field is always listed first in the Definefile window, followed by any synonyms that you have created for that field.

Defining a Group Field

A group field allows you to link together a series of fields under one name. A group field can also include selection and sort criteria. You can use this group field in R/LIST reports.

To create a group field, follow these steps:

1. Enter a name for the group field at the *Field name* prompt. The same conventions apply to group fields as to "F"-type fields.
2. After you have entered the name, the cursor will skip to the *Data type* prompt. Press [Shift-F7] to tell the Definefile window that you want to create a group field.
3. You will see a message that prompts you to confirm that you want to convert the field from an "F" to a "G"-type. Press [Enter] to confirm.

4. A Group window appears. Fill in the fields that constitute the group. You can also add selection and sort criteria for a report. Save your entries by pressing [F9].
5. When you return to the Definefile window, notice that the number at the *Pos* prompt disappears, since group fields are not associated with a record position. The rest of the prompts for that field will be skipped, since they do not apply to group fields.

Editing the Group

You can view or change the entries for a group field at any time. To do so, place the cursor on the group field you wish to edit, and press [Shift-F7].

Specifying Detail Information for a Field

The Definefile window allows you to fill in eight attributes for each field you define. In fact, there are several more attributes available.

Using softkeys, you can link from the Definefile window to two windows that allow you to define additional information about a field.

Dictionary Window

The Dictionary window displays 12 prompts for an individual field. All the prompts from the Definefile window are there, plus several more, including output format and validation patterns.

To link to the Dictionary window, put the cursor on the field for which you wish to edit the dictionary. Then press [Shift-F5].

Add or change any information you like about the field. When you are through, press [F9] to save your changes. Then press [Esc] to return to the Definefile window.

- ☛ For more information about dictionaries and field attributes, see the chapter “Advanced Revelation Dictionaries” in this section.

Dictionary Bonding Information Window

If you are working with a bonded file, there is quite a bit of extra information that you can specify for a field. This includes information such as the foreign field name, degree of acceptable data loss when reading or writing to the file, and the handling of nulls.

The place to do this is the Dictionary Bonding Information window. To access this window, put the cursor on the *Field name* prompt of the field you wish to edit. Then press [ShiftF4].

Fill in or change the prompts at the Dictionary Bonding Information window as required. Press [F9] to save your changes, and [Esc] to return to the Definefile window.

☞ For more information about the dictionaries and field attributes, see the chapter “Advanced Revelation Dictionaries” in this section. For background information on bonding concepts, see the chapter “Introduction to Bonding” in the *Environmental Bonding* section.

Advanced Revelation File Attributes

When you define an Advanced Revelation file (Linear Hash or ROS), you can specify a series of file attributes for the new file. By doing so, you can fine-tune the file to your application requirements and gain performance.

- ☞ In the vast majority of cases, the default values for the file attributes are perfectly adequate. The default attributes have been selected to present the best balance between speed and storage requirements.

Linear Hash File Attributes

The Linear Hash file type is self-optimizing in that it expands and shrinks to accommodate different quantities of data. When the file is “full” (per your definition — see “Resize threshold” below), the file is expanded and the data redistributed for optimal access. If you remove data, the file is shrunk, and the remaining data again redistributed.

This automatic resizing results in a very efficient file. In fact, for most applications you do not need to concern yourself with file attributes. However, under some circumstances you can override the default attributes and gain efficiency:

- If you are working with very large records (larger than 4K).
- If you know in advance approximately how many records you will have, and how large they are (on the average) *and* you are loading all the data at once. If so, you can pre-size the file and gain performance.
- If the network you are working on does not transfer data in 1K packets.

When you define a Linear Hash file, you can specify up to five attributes for the file. These are described below.

Number of records Enter the total number of records that you anticipate putting into the file.

- Average size** Enter the *average* (not largest) record size for your file. This number times the number of records tells Advanced Revelation how much data to expect for the file. It will then pre-size the file accordingly.
- Dictionary records** Enter the number of fields that you will be defining for the file.
- Frame size** By default, Linear Hash files store data in 1K “frames”. This is adequate for most data files. However, you may wish to specify a larger or smaller frame size for these reasons:
- You are using very large records (greater than 4K).
 - Your network uses a packet size different than 1K.
- In either case, you can match the frame size more closely to your situation. Possible values for frame size are 256 to 10,000 bytes.
- ☞ The larger the frame size you specify, the more memory your file operations will consume (six times the frame size). Use large frame sizes with discretion.
- Resize threshold** The Linear Hash filing system will attempt to keep your file about 80% full before expanding the size of the file. You can indicate a higher percentage if you wish to pack data tighter into the file (saving space). Conversely, a lower threshold value can result in faster access (on the average), but takes up more space.

ROS File Attributes

If you do not specify file attributes for a ROS file, Advanced Revelation will create a file that is optimized for 64K of data or less. If you will be storing more than 64K of data in a ROS file, use the file attributes prompts to create a more efficient file.

- ☞ You should not use ROS files if you will be creating large files (larger than about 256K). The efficiency of ROS files can suffer if the file is too large.
- ☞ Do *not* use ROS files to store data that will be shared on a network. If you will be sharing data with other users, you *must* use a Linear Hash file.
- Number of records** Indicate the number of records that you anticipate storing in the file.
- Average size** Estimate what the *average* (not largest) record size will be for your file. This number times the number of records tells Advanced Revelation approximately how much data you will be storing in the file. Advanced Revelation uses this information to determine how to optimize the distribution of data in the file.

Creating File Synonyms

You can create a synonym ("Qfile") for any file in your system. This is useful for two reasons. First, you can assign more convenient names to files. Second, a file synonym allows you to access files in a different application (account).

The Set File Command

To establish file synonyms, use the Set File window or command. To use Set File, you will need to know the name of the original file, and where it is located in the system.

If the original file is in another account, you will need to specify the name of that account. If that account is password protected, you will need to know the password before you create the synonym.

You have the option of specifying the name for your synonym. If you do not specify a name, Advanced Revelation will create a synonym simply called "QFILE". In this case, you can use the name "QFILE" whenever you would normally use a file name.

You can save your synonym permanently. Otherwise, the synonym name is available only for the current Advanced Revelation session.

To create a file synonym, follow these steps:

1. From the Advanced Revelation Main menu, choose the options **Tools-Files-Set File**. This displays the Set File window.
2. At the *Volume* prompt, enter the name of the volume on which the original file is located. For more information about volumes, see the chapter "Advanced Revelation Volumes" later in this section.
3. At the *Account* prompt, enter the name of the account to which the original file belongs. The default is the current account. If the account has a password, you must know the password to access the file using a synonym.
4. At the *File name* prompt, enter the name of the original file. Press [F2] to see a list of all files for the account and volume you have specified.
5. At the *Qfile name* prompt, enter the synonym name. If you do not indicate a name, the name "QFILE" will be assigned as the synonym.

☞ If you are already using the synonym name that you enter here, the synonym name will be changed to reflect your new definition.

6. At the *Option* prompt, enter "W" if you wish to save (write) your definition in a file. If you do not save your definition, the synonym name is only valid until you log off.

You can use the synonym file name as soon as you have finished defining it. If you save your synonym as a permanent name it will be accessible every time you log on to Advanced Revelation. However, you may need to attach the file before accessing it. For more information on attaching files, see the chapter “Advanced Revelation Volumes” later in this section.

Adding or Modifying Commands (the VOC File)

Advanced Revelation presents you with a very flexible environment — in fact, you can even change its entire vocabulary. Whenever you are in Advanced Revelation, there is a file that contains the entire vocabulary of commands that you can execute. This file is called the VOC file.

The VOC file contains an entry for each command you can execute at The Command Level (TCL). Because Advanced Revelation’s menus and windows typically execute TCL commands, the VOC file is also the basis for many system windows.

There is a separate VOC file for each application (account) that you have in your Advanced Revelation system. By modifying the VOC file for an account, you can have complete control over the commands that are available to the user. You can modify the VOC file to accomplish these tasks:

- Add new commands for the user. For instance, if you have created a procedure to post invoices, you can add a command such as POST to the VOC file.
 - Delete existing commands. If you wish to prevent a user from executing a specific command, you can remove it from the VOC file.
 - Create synonyms for existing commands. You can change the names of commands to suit your application or to match a system that the user is already familiar with. You can even translate commands into other languages if your application requires this.
- ☛ Remember that if you change the VOC file, you may be changing the way Advanced Revelation operates. Experiment with your changes carefully, and be sure to have a backup of your system.

Adding a New Command

When you create a new procedure, you can name the procedure and add the name to the VOC file. From then on, you can execute the procedure by using its name in a menu, at TCL, at the SQL window, or from within an R/BASIC program.

Your new procedure might take several forms. It may simply be a list or stack of commands to be executed in sequence, much like a batch file at DOS or OS/2. To create a batch command, you can use a special window available at the TCL prompt. For details, see “Introduction to TCL” in the *TCL* section.

You might also create a new command by writing an R/BASIC program. For example, as noted above, you might create an R/BASIC program that posts invoices, or one that creates a special report. When your program is finished, you can enter

its name in the VOC file by cataloging it. For details on creating and cataloging R/BASIC commands, see the “Introduction to R/BASIC” chapter in *R/BASIC*.

Deleting a Command

If you wish to prevent users from executing a command, you can remove it from the VOC file. You may wish to do this for security reasons, or because you have created alternative commands for your application.

To delete a command from the VOC file, follow these steps:

1. From the Advanced Revelation Main menu, choose the options **Tools-Records-Delete Record**.
2. At the *File* prompt, enter VOC.
3. At the *Record(s)* prompt, enter the commands you wish to delete. For example, to prevent users from changing their passwords, enter the command PASSWORD.
4. Press [F9] to execute the process.

- ✎ Removing a command from the VOC file does not remove any corresponding menu entries or windows. You must remove these separately by updating the menus (using the **Design-Menu** window), and by deleting or modifying the appropriate window in the COMMANDS file.

Creating a Synonym for a Command

If you wish to access a command under another name, you can create a synonym for it. To do this, copy the command to a new name. If you like, you can delete the old command at the same time. You can also leave the old entry in the VOC file, and make the command available under two (or more) names.

To create a synonym for an entry in the VOC file, follow these steps:

1. From the Advanced Revelation Main Menu, choose the options **Tools-Records-Recordcopy**.
2. At the *Source File* prompt, enter VOC.
3. At the *Destination File* prompt, press [Enter] to fill in VOC again.
4. At the *Source Record(s)* prompt, enter the name of the command for which you are creating a synonym.
5. At the *Destination Record(s)* prompt, enter the new name for the command.
6. If you wish to delete the old name of the command, enter “D” at the Option(s) prompt.
7. Press [F9] to create the synonym.

Transaction Processing

Advanced Revelation offers you a feature called transaction processing for protecting the integrity of your files. Transaction processing allows you to make complex, multi-file updates with complete assurance that all files and records involved in the update are made properly.

About Transaction Processing

In many applications, a single process must update several records. A simple example is a bank transfer. To transfer money from one account to another, you must debit the first account, and credit the second.

However, something might happen to prevent the files from being updated properly. One example is a power outage. A less dramatic but more common example is that another user on a network has a record locked that your application needs.

In either case, if only one of the operations took place, the files would not be synchronized. The result, of course, is a serious error.

Transaction processing prevents this by logically grouping all related updates into a single update operation or “transaction”. This transaction is then treated as one update. If any of the individual updates comprising the transaction cannot be completed, the entire transaction is cancelled. All the files that would have been affected by the transaction are left as they were.

In the banking example above, both updates would comprise a single transaction. If either account record could not be updated, the transaction would be cancelled. Neither account record would change, and the files would stay synchronized.

How Transaction Processing Works

Transaction processing stores all the updates you make in temporary files. Once you start a transaction, the results of any updates you make to files are assembled in temporary transaction files rather than being written to the actual data files.

When you are finished with a transaction, the data is copied from the temporary files to the data files. On the other hand, if you abort the transaction, the data is not copied to the data files, and they remain unaffected.

Any read operations that you include in your transaction will use the results of a preceding update — the record is read from the temporary file. This ensures that you see the most current version of the data. From your perspective, it is as if the changes had actually been made to the data files.

On a network or in a multi-user system, other users do not have access to your temporary transaction files. If those users read any records that you have updated during a transaction, but have not yet written to the data file, they will see an older version of the record.

Applying Transaction Control

You can use Advanced Revelation transaction processing with any files in your system. However, unless you are using SQL, you must add transaction control to the file or files to be monitored.

- Transaction control is applied automatically to files being accessed with SQL statements. If you have defined new files using the SQL CREATE TABLE command, transaction control is already in effect for those files.

To add transaction control to a file or a group of files, use the TCL command CONTROL_ON. Refer to the “TCL Command Reference” chapter of *TCL*.

It is not necessary to add transaction control to all files involved in a transaction. If the updates you are making to a file are not critical to the integrity of a transaction, you do not need to add transaction control to that file.

- Transaction processing adds slight overhead to the file while monitoring transactions. If performance is a concern, do not add transaction control unless you require it for your application.

For networks or multi-user systems, transaction control also enables you to restrict the maximum number of locks available to a workstation (see “Locking and Unlocking” under “Managing Transaction Processing” below).

Domain Controls

When you apply transaction control to a file, you are also installing conversion and validation controls for data. These are called “domain” controls, and have effect only for cursor input and output in R/BASIC.

- For more information, see the chapter “Programming with R/BASIC” in *R/BASIC*.

Checking for Transaction Control

To determine if transaction control has already been added to a file, use the command LISTMEDIA plus the appropriate volume name at TCL. If the word “PROTECT.MFS” appears in the column “MFS” for a file, transaction control has been added. If nothing appears in the MFS column, no transaction control has been added.

An alternative method is to use the TCL commands CONTROL_ON and CONTROL_OFF. If you enter CONTROL_ON at TCL with no file name parameters, you will see a popup of all files in the volume that do not have transaction control already installed. Conversely, if you enter CONTROL_OFF with no file names specified, you will see a popup of files that already have transaction control installed.

The popups displayed by the CONTROL_ON and CONTROL_OFF commands are multi-selection popups. These allow you to change whether transaction control is installed

for a file by pressing [Enter]. If you are using the commands only to report the status of transaction control for a file, be careful not to make changes.

Removing Transaction Control

You can remove transaction control from a file at any time. To do so, use the `CONTROL_OFF` command. Refer to the “TCL Command Reference” chapter of *TCL*.

You can remove transaction control from SQL files with the `CONTROL_OFF` command; however, when the files are accessed again using an SQL statement, transaction control will be reapplied.

Using Transaction Processing

With Advanced Revelation’s transaction processing, you have complete control over how and when transactions are processed. You can initiate and complete a transaction using simple commands.

Transaction processing consists of these steps:

- Initiating (starting) a transaction.
- Making all the updates required for the transaction. Your changes are stored in temporary transaction files.
- “Committing” the transaction to indicate that all updates have been made. This copies the data from the temporary transaction files to the data files.

If all goes well, these are the only steps required. However, an additional process is available if you detect a problem before you have committed your transaction:

- “Rolling back” (cancelling) a transaction. This clears the temporary transaction files.

To control transaction processing, use the `TRANSACTION` command. This command can be entered at the TCL window, or can be executed from within an R/BASIC program with the `EXECUTE` statement.

- ☛ For more information about the `TRANSACTION` command, see the “TCL Command Reference” chapter of *TCL*. For more information about programming in Advanced Revelation, see *R/BASIC*.

Starting a Transaction

To enable transaction processing, use the statement `TRANSACTION ON`. Once a transaction has been started, all updates are monitored, whether you update files using a window, an R/BASIC program, or any other process.

- ☛ Remember that transaction processing only monitors files with transaction control in effect. See “Applying Transaction Control” above.

There is no predetermined size or duration to your transaction. Any operations you specify from the time you start the

transaction until the time you end it are considered to be a part of the same single transaction.

Starting Transactions Automatically

Using the *Auto-Transaction Start* setting at the Concurrency Control Environment window, you can have transaction processing automatically enabled for all files with transaction control.

Display the Concurrency Control Environment window by choosing the **Management-Environment-Concurrency** options from the Advanced Revelation Main menu.

Automatic processing is then active after you log in or reset the system, and after each commit or rollback operation.

Committing a Transaction

When you have finished making all the updates for your transaction, you must close out the transaction. This process is called “committing” your transaction, because you are then committed to the updates you have made.

To commit a transaction, use the statement TRANSACTION COMMIT.

Rolling Back a Transaction

You may occasionally determine that your transaction is not proceeding properly, or that you made an error during an update. If so, you can cancel the transaction, and roll back the changes made by the transaction.

To cancel a transaction, use the TRANSACTION ROLLBACK statement.

The data files that were to be updated by your transaction remain in their original state because your updates are never copied from the temporary transaction files to the data files.

Pending Transactions

If you forget to commit your transaction, when you log off or reset your system, Advanced Revelation will prompt you to commit or rollback the transaction. When you reset you also have the option to ignore the transaction.

After the transaction is committed or rolled back, any locks held by your workstation are released. If you choose to ignore the pending transaction when resetting, the locks are not released and the transaction remains active.

Recovering from a System Failure

If your system fails or is shut down *after* you have already committed a transaction, the updates you have logged to the temporary transaction files might not have finished copying to the data files.

- ☛ If you have not yet committed a transaction when your system fails, all changes you have logged are lost, and the data files remain in their original, unaltered state. You can only recover transactions that were committed before the system failure.

When you log back into Advanced Revelation after a system failure during the commit process, a message displays telling you that there is an unfinished transaction. You have three choices at this point:

- Finish the commit process that was interrupted.
- Ignore the transaction, and carry on with your work.
- Delete the transaction. This deletes the data in the temporary transaction files, and effectively rolls back any portion of the transaction that was not completed.

Committing an Unfinished Transaction

If you choose to commit the transaction, Advanced Revelation will attempt to finish the commit procedure. The transaction process will restart the commit process, copying the data from temporary files to the data files.

Ignoring an Unfinished Transaction

You can have Advanced Revelation ignore the transaction that remains unfinished when you log back in. You might wish to do this, for instance, if you had been updating files that are not available at the time you log back into Advanced Revelation.

If you choose to ignore an unfinished transaction, you can complete it later using the TRANSACTION RESTART command. This command is a manual version of the procedure that Advanced Revelation uses when it detects an unfinished transaction at logon.

As a rule, you will want to finish a transaction as soon as possible. The changes that you have logged for a file will not be available to you or to other users until the transaction is complete. Other processes or other users may be dependent on changes that you have been logging to a file.

In addition, your unfinished transaction may be holding up transactions that other users have logged. Under some circumstances, transactions are queued for processing. If your transaction is at the top of the queue, other transactions must wait until yours is completed.

- ☛ Transaction queues are described under “Managing Transaction Processing” later in this chapter.

Deleting an Unfinished Transaction

You may decide when you log back on that it is better to simply clear the unfinished transaction. For instance, you may determine that is more appropriate to finish making file updates manually.

If you choose to delete an unfinished transaction, records that had not yet been committed when the system failed are removed from the temporary transaction files. The system also removes control information indicating that you were in the middle of a commit procedure.

- ✎ The delete option does not remove changes that had already been successfully made when the commit process was aborted. If your system failed and the transaction was partially complete, the data files will be partially updated. Choosing the delete option may therefore result in a loss of data integrity.

Managing Transaction Processing

The information that follows is intended primarily for advanced users and system administrators. When you use transaction processing, you should be aware of how Advanced Revelation is monitoring and protecting your system. Not only will this help you when you have problems, but it will help you coordinate with other users in a networking or multi-user environment.

Assigning a Transaction File Volume

The temporary files used for your transactions need to be stored in a single volume. The default volume is TRANSACT. To assign a different volume for these files, use the *Transaction File Volume* prompt at the Concurrency Environment window.

To display the Concurrency Control Environment window, choose the **Management-Environment-Concurrency** options from the Advanced Revelation Main menu.

Phases of the Commit Process

Once you commit your transaction, the records you have updated are copied from the temporary transaction files to the actual data files. As they are written successfully to the data files, they are removed from the transaction files.

At this stage, the commit process is only minimally vulnerable to system failure. There are three reasons it is virtually guaranteed to be successful:

1. The updates only begin when you have signalled that the transaction is done. This means that you are confident that the updates you have made are correct and complete. In addition, it is assumed you will not be halting a transaction because of an error that you detect, or because you were unable to access a record you needed.
2. The update of the data file from the transaction file happens in a burst. Because all the updates are concentrated into a very short time, the chances of

a system failure during the course of the commit process are reduced.

3. The transaction processing system maintains a log of the updates it makes during a commit process. If the commit process is interrupted (by a workstation or system failure, for example), the log allows Advanced Revelation to restart the update process when you next log on (see “Restarting a Transaction” above).

Using the Commitlog

When each transaction is committed, user and workstation information is written to a “commitlog”. This is a central record of all transactions currently being committed.

All users can display the commitlog. If you are in the SYSPROG account, however, additional functions are available.

If you are in the SYSPROG account when you display the commitlog, the commitlog is locked, preventing committed transactions from being executed. This is necessary to give the system administrator an accurate representation of current commits.

If *Commit Protection* is enabled (see “Queueing Committed Transactions”, below), you are able to remove entries from the queue if necessary.

Assigning a Commitlog Volume

The file used to maintain the commitlog needs to be stored in a centralized volume. The default volume is TRANSACT. To assign a different volume for these files, use the *Commit Log Volume* prompt at the Global Environment window. Choose the **Management-Environment-Global** options from the Advanced Revelation Main menu to display the Global Environment window.

- ☛ All users must be accessing the same commitlog volumes. If you change the volume for the commitlog, be certain that all users are able to attach this volume.

Displaying the Commitlog

To display the commitlog, use the TCL command COMMITLOG.

When a commit queue is enabled (*Commit Protection* is set), the commitlog displays:

- Users with currently committed transactions.
- Workstation identification for each user.
- Data files having a commit queue.

- Transaction entries in the commit queue for each file.

If a commit queue is not enabled (*Commit Protection* is not set), the commitlog displays:

- Users with currently committed transactions.
- Workstation identification for each user.

Queuing Committed Transactions

On networks or multi-user systems, there is a good possibility that more than one user will commit a transaction to a file at the same time. To ensure data integrity, you can choose to protect the orderly execution of committed transactions by enabling queuing. Each transaction is then assigned an entry in a queue for the target file(s) as it is committed.

The *Commit Protection* setting at the Global Environment window enables queuing for transaction updates, and thus assures you of the highest level of data integrity. Choose the **Management-Environment-Global** options from the Advanced Revelation Main menu to display the Global Environment window. The default setting for *Commit Protection* is "Yes".

- ☞ Queued transactions are guaranteed only in SQL with a *Consistency Level* setting of 4 and *Commit Protection* set. For more information, see the chapter "Managing the System" in *SQL*, and the *Networking* section.

When you do not need the measure of data integrity provided by queuing transactions, disabling *Commit Protection* will improve performance considerably.

Removing Users from the Commit Queue

When committed transactions are queued, each transaction *must* be executed in order to execute any subsequent transactions in the file queue. For example, if a system failure should occur at a workstation with a transaction in a queue, any remaining transactions in the queue would be unable to execute.

In such emergencies, it may be necessary to remove a transaction entry from the queue. Doing so, however, may risk data integrity by forfeiting the order of execution for commits to the file. A subsequent transaction may access invalid data because of the missing entry.

Only a user in the SYSPROG account can remove a transaction entry from the commit queue. To do so, use the TCL command COMMITLOG to display the current commitlog.

- ☞ For more information about COMMITLOG, see the "TCL Command Reference" chapter of *TCL*.

When the commitlog is displayed, press [F6] for a popup of the softkeys. There are two operations available:

1. Delete entry from commitlog:

To remove a transaction entry for a specific user from the file queue(s), position the cursor at the appropriate user name and press [Shift-F1].

To remove the first transaction entry in a file queue, position the cursor at the appropriate file name and press [Shift-F1].

This function is disabled if *Commit Protection* is off.

2. Clear entire commitlog:

To remove all transactions entries from all file queues, press [Shift-F2].

In both cases, the user's transaction files are not affected. The entry is simply removed from the commit queue for a file. This enables the user to subsequently restart a transaction, using a TRANSACTION RESTART statement.

Locking and Unlocking

If you are working in a network or multi-user environment, generally you will be locking records before updating them, and unlocking them after the update is finished. During transaction processing, the behavior of these operations is altered slightly in order to guarantee the integrity of the files.

When you issue a lock during a transaction, the lock is passed through the transaction monitor and set on the network (assuming that the record is available). The subsequent unlock command, however, is *not* passed through to the network. Instead, it is held by the transaction monitor until you commit your transaction.

This is because, as with any update, the records in a transaction remain locked until they have been written to a file. Because the changes during a transaction are still in temporary files, the locks are not released until these changes are written to the data file.

There are two problems you might encounter:

1. You might be keeping other users out of records for long periods until you commit your transaction, even though within your transaction you are immediately unlocking records.

2. You might use up a great many locks on your network. Some networks can only maintain a limited number of locks concurrently, and the accumulated locks you set during your transaction may cause the system to exceed this limit.

To counteract these potential problems you can:

- Make your transactions smaller, or perform commits or rollbacks more frequently.
- Eliminate needless locking on non-shareable files.
- Use file locking where possible, to conserve available locks.

Unlocking All at Commit

You can choose to release *all* locks when you commit a transaction. This includes any locks set by an Advanced Revelation environment or process (such as the Editor or Paint), as well as locks that you might have set yourself.

To release all locks, set *Unlock All at Commit* at the Concurrency Control Environment window. Using this setting will increase performance considerably for large transactions.

☞ For more information, see the *Networking* section.

Limiting Available Locks

Applying transaction control also enables a lock limiting function. If you wish to conserve or control the number of locks on your network or multi-user system, you can set the maximum number of locks available to a workstation.

The *Lock Limit* setting is available at the Concurrency Control Environment window. To display the Concurrency Control Environment window, choose the **Management-Environment-Concurrency** options from the Advanced Revelation Main menu.

Setting *Lock Limit* to zero removes workstation limits on the number of locks available. This is the default setting.

☞ Most networks restrict the total number of locks that can be set. Refer to your network documentation.

Assessing Transaction Status

You can monitor the status of transaction processing at your station by using the TRANSACTION STATUS statement. This produces a report of the following:

- A list of files affected by the current transaction.
- The starting date and time for the current transaction.

Advanced Revelation System Files

Advanced Revelation provides system files for every account. These files are used to store application windows, reports, forms, menus, popups, lists, and so on. System files generally reside on the REVBOOT volume, rather than in a separate volume with the data and dictionary files. If you transfer an application to another system or network, you will need to copy all the system files as well as the data and dictionary files.

Application System Files

The following files are associated with all Advanced Revelation accounts. Each time you create part of an application, such as a window using Paint or a list using R/LIST, Advanced Revelation writes a record to the appropriate system file shown in the table below.

System File	Contains
CAPTURED	captured keystrokes.
FILTERS	filtered templates.
HELP	developer's help records.
IMPORT-EXPORT	import/export processing templates.
LABELS	mailing label templates.
LISTS	saved lists of record keys.
MACROS	macro templates.
MENUS	menu templates.
MERGES	merge processor templates.
POPUPS	popup templates.
QFILES	names of synonym files.
QUERIES	R/LIST templates.
REPORTS	form templates.
SQL_OBJ	generated source and object code for SQL Modules.
TEMPLATES	window templates.
UPDATE	batch update templates.
VIEW_CATALOG	information about SQL Views.
VOC	(vocabulary) words and keywords used as TCL commands or command modifiers.

Global System Files

Advanced Revelation contains files which do not belong to any one account. These are global files, and available to the system regardless of which account you are in. Advanced Revelation's global system files contain programs or data which are used in all applications.

Global File	Contains
ASSIST.SQL	control records for the SQL Assistant.
ASSIST.SQL CONTROL	control records for the SQL Assistant.
COMMANDS	templates for command windows.
INCLUDE	\$INSERT records for R/BASIC programmers.
MESSAGES	system messages.
MRG.PRINT-CONFIG	printer driver information for the merge processor.
SYS.HELP	system help.
SYS.MENUS	system menus.
SYS.POPUPS	system popups.
TYPE_MAPS	mapping information for foreign data types.
TYPE_TABLE	definitions for Advanced Revelation data types.
VERBS	system programs.

Advanced developers can edit global files and remove records to restrict users from performing certain operations. However, if you change global files, you are changing the fundamental way in which Advanced Revelation works, and you might not be able to guarantee that your application will conform to the documentation, or that it will be compatible with the next release of Advanced Revelation.

The SYSTEM File

Advanced Revelation uses a special system file called SYSTEM on the SYSPROG account for storing information about users and accounts. The SYSTEM file includes user, account, video, and environment records:

User records specify an account, a set of video attributes, an environment, and user security information for each user on the system.

Account records specify a set of video attributes, an environment, and account security information for each account defined on the system.

Video records contain the video (color) settings associated with each user defined on the system.

Environment records specify the characteristics defined for each user's environment on the system.

When you create a new account or user on the SYSPROG account, or define new video or environment settings for a user or account, new records are written to the SYSTEM file. An account or user must have a record in this file in order to be recognized by Advanced Revelation.

Several additional records contained in the SYSTEM file are reserved by Advanced Revelation. Do not change or delete these special records.

Chapter E 3

Advanced Revelation Dictionaries

Understanding how to use dictionaries is one of the keys to maximizing the power of Advanced Revelation. This chapter explains the importance of the dictionary. It tells you about the information contained in the dictionary and describes how to create a data dictionary and build a dictionary structure. It also explains the relationship of the dictionary to other processes such as Paint and R/LIST.

About Advanced Revelation's Dictionaries	E 3 . 1
Creating a Data Dictionary	E 3 . 1
Dictionary Record Types	E 3 . 3
Creating Temporary Fields for Reports	E 3 . 6
Paint Autobinding	E 3 . 7
Available Dictionary Reports	E 3 . 7
Dictionary Characteristics and Record Format	E 3 . 8



About Advanced Revelation's Dictionaries

Each file in Advanced Revelation (including both system and data files) has an associated dictionary file. The dictionary defines the format and layout of the fields in the data records.

Dictionaries are used for much more than just field definitions, however. Because dictionary records may reference user-defined formulas or routines written in R/BASIC, you can access virtually every feature and command in Advanced Revelation through the use of dictionaries.

Reports

All report processes rely on the dictionary:

- R/LIST commands are dictionary-driven. R/LIST requires the file dictionary both to identify which fields to display as well as how the display should appear.
- Forms, Labels, and Merge reports are defined using fields from the dictionary of the file being accessed.

Paint

Paint uses or defines dictionary entries as you construct data entry windows.

SQL

SQL commands are dictionary driven. SQL requires the dictionary to identify which columns to display as well as how the display should appear.

R/BASIC

Dictionary definitions are often used in R/BASIC application programs or other dictionary records, especially when you are defining formulas for symbolic fields.

Creating a Data Dictionary

All of the Advanced Revelation tools used to define files will create and update a dictionary file as part of the file definition process. These are the tools that create a dictionary:

- The Dictionary window. Information about this window appears below.
- The Definefile window. Details about this window appear in the "Managing Files and Records" chapter.

- The **DEFINEFILE** command at The Command Level (TCL). For information about the TCL **DEFINEFILE** command, see the chapter “TCL Command Reference” in the *TCL* section.
- **Paint**. This Advanced Revelation tool allows you to create a dictionary defining a new file while you are painting a new window. Use of **Paint** is discussed in the *Developing an Application* section.
- The SQL command **CREATE TABLE**. For details on using this command, see the *SQL* section.

Data and Dictionary Files

The file definition process consists of:

- Creating the dictionary of the file.
- Establishing field definitions for the new file.
- Creating the data portion of the file.

The dictionary portion of the file contains field definitions that describe the data in the file. The data portion of the file actually stores the data records. Advanced Revelation determines how to display the data in the data file, based on the field definitions in the dictionary.

The dictionary portion of a file is given the name of the file prefixed with “**DICT.**”. For example, the file named **CUSTOMER** would have a dictionary named **DICT.CUSTOMER**. Advanced Revelation allows records to be read from and written to the dictionary file just as if it were a data file. For example, the **R/LIST** command **LIST CUSTOMER** would list records from **CUSTOMER**, while the command **LIST DICT.CUSTOMER** would list records from the dictionary of **CUSTOMER**.

☞ You can also reference the dictionary file by replacing the period separating **DICT** and the file name with a space. In the example above, **DICT CUS-
TOMER** is equivalent to **DICT.CUSTOMER**.

Since the data file and its dictionary have different names, you can manipulate them separately. This is sometimes useful. For example, although you would normally use **RECORDCOPY** to copy both the data and dictionary of a file, it is possible to copy only the data (for use with another customized dictionary). You can also delete the dictionary of a file without affecting the data records, or delete the data without affecting the dictionary. Be careful when manipulating only the dictionary or data of a file; many features of Advanced Revelation will work improperly if the dictionary of a file is missing.

Building a Dictionary Structure

After a dictionary file is created, you can use the Definefile window to define fields in the data dictionary. Alternately, Advanced Revelation will define fields automatically as you build a window with **Paint**. This is referred to as “Autobinding” in **Paint**.

Automatic dictionary construction in Paint is suggested, particularly for simple applications. See “Paint Autobinding” below for more information. Also see the chapters called “Creating Windows with Paint” and “Adding Advanced Features to Windows with Paint” in *Developing an Application*.

If you want to define a bonded file, such as a dBASE or an ASCII file, the recommended tool is the Definefile window. The Definefile window enables you to create the dictionary, establish field definitions, specify any special file attributes, and create the data portion of the file. To access the Definefile window from the Advanced Revelation Main menu, select **Tools-Files-Definefile**. See “Managing Files and Records” for details on the operation of this window.

Modifying Existing Field Definition

If you wish to modify an existing field definition in the dictionary, use the Dictionary window. Using the Dictionary window, it is easy to review all information about a field. To access the Dictionary window from the Advanced Revelation Main menu, select **Tools-Files-Dictionary**. The online help keys [F1] (Context) and [Ctrl-F2] provide details on the operation of this window.

- ✎ You can also use the Dictionary window to create the dictionary portion of a file. Then use another tool — Quick Paint, the Definefile Window, or the MAKEFILE command at TCL — to create the data portion of the file.

Dictionary Record Types

Each dictionary definition is stored as a record in the dictionary file. The first field of each dictionary record holds the letter code which defines the field as being one of three dictionary types:

- F Field
- S Symbolic
- G Group

The Dictionary Types determine how the value for a dictionary field is calculated when you reference the field in R/LIST or some other process.

The Field Dictionary Type F

Most of the field definitions in the dictionary are for “F”-type fields. As the name would suggest, an “F”-type field definition defines the characteristics of individual data fields. The value of an “F”-type field is generated by simply extracting a particular field from the record.

Consider the following sample format for the records in a simple CUSTOMER name and address file:

Field 1 Number
 Field 2 Last Name
 Field 3 First Name
 Field 4 Middle Initial
 Field 5 Address 1
 Field 6 Address 2
 Field 7 City
 Field 8 State
 Field 9 Zip Code
 Field 10 Contact Dates

For this sample file, there would be at least 10 field dictionary records, one for each of the fields in the record.

Most fields are single value fields. That is, they each hold a discrete piece of information. Note, however, that in the example, field 10 (Contact Dates) is designed to hold more than one piece of information (more than one date). This kind of a field is known as a multivalued field. For more information on multivalued fields and how they are used, see the “Advanced Revelation Files and Records” chapter in the *Filing Systems* section.

The Symbolic Dictionary Type S

Symbolic dictionary fields (usually called “symbolic fields”) produce values calculated using formulas. These formulas may be simple, or they may be very large and practically unlimited in complexity. When you use the Dictionary window or the Definefile window to construct symbolic fields, the Formula window is displayed to help you enter the desired symbolic formula.

- ☛ You should have some familiarity with R/BASIC when constructing formulas. Refer to the *R/BASIC* section for more information. Also, see the “Building Formulas” chapter in *Developing an Application*.

Symbolics may be used in Advanced Revelation for a number of purposes. These include:

Synonyms for Fields Probably the simplest use of a symbolic field is to generate synonyms for other fields. For example, the field CUSTOMER_NUMBER could also have a symbolic synonym CUST_NO, consisting of the CUSTOMER_NUMBER field. The formula would be:

```
@ANS = {CUSTOMER_NUMBER}
```

You could then reference the data in the field using both the name CUSTOMER_NUMBER and the name CUST_NO. You can define multiple synonyms for any field.

Calculations Based on Record Data

Symbolic fields are often used for field or record data calculations.

The symbolic formula @ANS=LEN((CUSTOMER_NUMBER)) would be used to calculate the length of the field CUSTOMER_NUMBER.

The formula @ANS=LEN(@RECORD) would be used to calculate the length of the entire record.

Data Constants

You can construct a symbolic field to contain any constant numeric or text value. For example, the formula:

```
@ANS = 3.14159265
```

(an approximate value of pi) will assign the constant value of pi to the symbolic field. The formula

```
@ANS = "ABC"
```

will assign the constant text value "ABC".

Data Manipulation and/or Translation

Examples of data manipulations you can perform with symbolic fields are lower to upper-case text conversion and ASCII/EBCDIC text translation.

Data Formatting

Symbolic fields can be used to format data from other fields or to add literals to data on file. For example, the formula:

```
@ANS = {CITY}:' , ':{STATE}:' ':{ZIP_CODE}
```

would be used to produce the last line of a typical mailing label.

Joining Files

You can access data from any Advanced Revelation file (subject to security considerations). The formula:

```
@ANS = XLATE("CUSTOMER","100",9,"X")
```

would extract the ZIP code for customer number 100 from the example CUSTOMER file.

The Group Dictionary Type G

Use Group types to reference a group of fields or R/LIST options by a single name. For example, you might define the group field ADDRESS to connect the fields ADDRESS_1, ADDRESS_2, CITY, STATE and ZIP_CODE in the example CUSTOMER file. Whenever you wished to display this group of fields, you could simply reference the Group field ADDRESS.

You could also define another group STANDARD_SORT (consisting of the text BY ZIP_CODE BY LAST_NAME BY FIRST_NAME) as the standard sort order for a report. Simply include the field STANDARD_SORT in a report (for example, LIST CUSTOMER STANDARD_SORT) whenever you want the report sorted according to these fields.

The Default Group Fields

There are two standard group fields for every file used by Advanced Revelation — the @CRT group, and the @LPTR group. These are used as default groups for reports.

The @CRT group specifies what fields or R/LIST options will be used as default specifications for reports displayed on the screen if no other fields are specified for display. For example, the simple TCL command LIST VOC will use whatever specifications exist in the @CRT record (if any) as a default. However, the TCL command LIST VOC F1 will ignore the @CRT record, since the specification of the field F1 is present. The @CRT record is automatically created when the file dictionary is created; it is blank at that time.

The @LPTR group (like the @CRT group) specifies what fields or R/LIST options will be used as default specifications for reports, except that it is used for printed reports. For example, the simple TCL command LIST VOC (P) will use whatever specifications exist in the @LPTR record (if any) as a default. The @LPTR record is not built along with the data dictionary; it must be created if desired.

Creating Temporary Fields for Reports

You can create temporary dictionary expressions from the TCL window when generating R/LIST reports. Press the Expressions softkey [Shift-F3] while at the TCL window to display the Expression window. The Expression window enables you to define a Symbolic field while entering an R/LIST command at TCL. You can use the Expression window to create symbolic fields as they are needed, without exiting the TCL window.

Fields created with the Expression window are temporary. They are stored in the file dictionary with the name "EXP" prefixed to the relative expression number. For example, if two temporary Symbolic fields are created for use in an R/LIST query, they will be stored in the dictionary as EXP1 and EXP2. They remain until overwritten by later R/LIST queries.

- ✎ Do not build regular, permanent dictionary records with the name EXP nnn (where nnn is some integer), since they may be destroyed through use of the Expression window.

Paint Autobinding

When you use Paint to design an application window, you must specify a binding method to be used. The binding method determines the way the prompts in the window relate to the dictionary of the file being accessed.

You should use Autobinding (the default in most cases) if the dictionary of the file does not already exist when the window is built. You would normally use this method when defining a new file as the window is being designed. When autobinding is active, Paint creates a new dictionary record for each prompt field.

Autobinding is the recommended method of building dictionaries for new users of Advanced Revelation, or for simple applications. If necessary, you can always customize dictionaries later with the Dictionary window.

- ✎ For more information about available binding methods in Paint, see “Adding Advanced Features to Windows with Paint” in *Developing an Application*.

Available Dictionary Reports

Advanced Revelation provides two convenient ways to list information contained in the dictionary.

Listing the Contents of a Dictionary

You can use the List Dictionary window to display or print a report of the contents of a file's dictionary. To access the List Dictionary window from the Advanced Revelation Main menu, choose **Tools-Files-List Dictionary**.

The report contains information about the basic characteristics of each field in the file:

- Field name
- Dictionary type
- Position in the record
- Master flag
- Single/Multivalue
- Data type
- Justification
- Length

Listing Information Using the SYSCOLUMNS Table

SQL users can use the SYSCOLUMNS table to view dictionary definitions for all of the columns in a table. R/LIST tools, including EasyWriter, can also use the SYSCOLUMNS file to obtain selected information about the fields in all files that are attached.

If you use the LISTDICT command at TCL to list the SYSCOLUMNS file, you will see that the file has 38 fields. These fields correspond to the first 38 positions in the dictionary record format. For a brief explanation of the purpose of each field in SYSCOLUMNS, see “Dictionary Characteristics and Record Format” at the end of this chapter.

Dictionary Characteristics and Record Format

You can specify a number of different dictionary characteristics for each data field, ranging from simple to complex. Simple characteristics that are usually defined for every field in a file include:

- The field name.
- The display length, which specifies how much of the field should be displayed on a report, regardless of the actual length of the field data.
- The field justification, which specifies whether the data should be left or right justified when displayed.
- The column heading name that is to appear on reports.

Complex characteristics you might use less frequently include:

- Validation pattern, which checks the validity of data entered by the user.
 - Output format, which “translates” the field into a more readable format when displayed. An example would be a monetary conversion.
 - Formula (for a symbolic field) to perform special data manipulation.
- ☛ For more information on specifying validation patterns and data types in the dictionary, see “Data Validation and Formatting” in the “Adding Advanced features to Windows with Paint” chapter in *Developing an Application*.

Dictionary Record Format

To display an Advanced Revelation dictionary record in the Editor, you can enter a TCL command with the following syntax:

```
EDIT DICT.filename fieldname
```

All Advanced Revelation dictionary records have the same physical format, as follows:

Position 1 Dictionary Type	The Dictionary Type specifies what kind of field it is: F=Data Field; S=Symbolic; G=Group. The type indicates how the field is used.
Position 2 Position	The Position specifies the field's place in the record. The first field in the record is position 1, the second field is position 2, etc. The position number must be an integer from 0 through 65530. A 0 indicates that this is a key field.
Position 3 Column Heading	The Column Heading specifies the text that will display as a column heading on reports. In R/LIST reports, the column heading is always displayed in its entirety, regardless of the field Display Length specification (see Display Length below). Therefore, you should make sure that the column heading text is not longer than the field's Display Length.
Position 4 Single/Multivalued	This specification determines whether the field will contain a single-value (S) or will contain multiple (M) entries (repeating data).
Position 5 Key Part	For key fields, the Key Part specification allows you to create a multi-part key. Key Part specifies what part of the record key this field makes up. This should be 0 or null for files with single-part keys.
Position 6 Indexed	The Indexed specification is a flag that indicates whether the field is indexed. This information is protected. It can only be modified by using the Indexing windows to add or remove an index for the field.
Position 7 Output Format	The Output Format specification allows you to control how data stored in this field is formatted for display in reports and windows. Some data, such as date and time, is converted to an internal format when it is stored in the field. In this case, an output format enables you to convert the stored data so that it displays in a recognizable format.
Position 8 Formula	For "S"-type (symbolic) fields, the Formula specification stores the lines of R/BASIC code to be used for the field formula. Each line is a value in a multivalued field.
Position 9 Justification	The Justification specification determines how data is positioned when it is displayed. In addition, for fixed length file types like dBASE and ASCII, justification determines how the data is positioned within the field when it is stored. L=Left; R=Right; C=Centered; T=Text (displayed exactly as entered, all formatting retained).

Position 10 Display Length	The Display Length specifies the default number of characters to display for the field. Data exceeding this length will be truncated on the display when it is output. See Column Heading (position 3) above.
Position 11 Validation Pattern	The Validation Pattern specifies any desired pattern match, range check, etc. to be used to verify the validity of the field data when it is entered. Validation patterns may also convert date, time, and decimal number data to internal format for more efficient data manipulation.
Position 12 Data Type	The Data Type specifies the kind of information that is stored in the field. The data type is identified by name (INTEGER, CHAR, DATE), followed by any special attributes that determine the characteristics of the field.
Position 13 Source	The Source specification provides documentation regarding the source of the data. To enter a Source specification, you must edit the dictionary record in the Advanced Revelation Editor.
Position 14 Description	The Description specification provides any information you choose to enter about the field. This information can be displayed as online help (using an HD code).
Position 15 Foreign Type	For fields in bonded files, the Foreign Type specification provides the name of the field's data type as the foreign environment would refer to it. For example, in a dBASE file, a field with the data type of VARCHAR would specify M (dBASE Memo) as its foreign type.
Position 16 Foreign Name	For fields in bonded files, the Foreign Name specification indicates the name by which the field will be known in the foreign environment.
Position 17 Reserved	This position is reserved for use by Environmental Bonds.
Position 18 Foreign Attributes	For bonded files, the Foreign Attributes specification defines precisely how the data in a field should be read for use in Advanced Revelation, and how it should be written to the file. The attributes specified at this position are bond-specific. All data types define a set of default foreign field attributes that can be modified.
Position 19 Reserved	This position is reserved for use by Environmental Bonds.

- Position 20
Null Truncation** For bonded files, the Null Truncation specification establishes null handling for the field. A Y indicates that the bond will accept null data for this field. N or no value indicates that the bond will not accept null data for this field. The second value at this position specifies a character or literal string that will be used in Advanced Revelation to identify a null value in the field. For more information, see the "Introduction to Bonding" chapter in *Environmental Bonding*.
- Position 21
Reserved** This position is reserved for use by the Advanced Revelation indexing system.
- Position 22
Cross Referenced** The Cross Referenced specification indicates that there is a cross reference index maintained for this field. The value at this position is the name of the symbolic field that calculates the cross reference index (using a formula that accounts for the delimiters and stop list specifications).
- ☞ The value at this position is maintained by the Advanced Revelation indexing system. It can only be modified by using the Indexing windows to add or remove a cross reference index for the field.
- Position 23
Relational Index** The Relational Index specification indicates that this field is used to update a relational index in another file. The field supplies the key to a record in the other file.
- ☞ The value at this position is maintained by the Advanced Revelation indexing system. It can only be modified by using the Indexing windows to add or remove a relational index for the field.
- Position 24
Related** The Related specification indicates that the field contains a relational index. The indexing system updates the field with keys to related records that are in another file.
- ☞ The value at this position is maintained by the Advanced Revelation indexing system. It can only be modified by using the Indexing windows to add or remove a relational index that updates this field.
- Position 25
Protected** The Protected specification is a flag that determines whether the data in this field can be modified. The relational indexing process sets the flag to 1 to indicate that the field stores a relational index. In this case, only the indexing system can update the field.
- ☞ The value at this position is maintained by the Advanced Revelation indexing system. It can only be modified by using the Indexing windows to add or remove a relational index that updates this field.

Position 26 Lowercase Index	For indexed fields, the Lowercase Index specification determines whether the indexed values are converted to uppercase or are indexed as entered (upper/lowercase). 0=convert to uppercase; 1=maintain as entered (upper/lowercase).
Position 27	This position is reserved for use by the Advanced Revelation indexing system.
Position 28 Master Flag	For bonded files, the Master Flag specification determines whether or not this field definition is used by the bond when it reads and writes to a file. A specification of 1 indicates that this is the "master field". The bond always uses the master field definition when it reads and writes data to the file. A specification of 0 or null indicates that this is a synonym for the master field. A synonym field can have a different description of the data.
Position 29 Default	For SQL tables, the Default specifies a default value for the current column. When a row is inserted with no value specified for this column, the Default specification is used.
Position 30 Reference	For SQL tables, the Reference specification indicates that this column is a foreign key to the listed table(s).
Position 31 Null	For SQL tables, the Null specification is a flag that indicates whether the column must be assigned a non-null value. 0=column may be assigned a null value; 1=column may not be assigned a null value.
Position 32 Unique	For SQL tables, the Unique specification is a flag that indicates whether or not the column may contain any duplicate values. 0=column may contain duplicate values; 1=for all rows in the table, column must contain a unique value.
Position 33 through 38 Reserved	These positions are reserved for use by Advanced Revelation.
Position 39 Input	For bonded files, the Input specification sets the degree of acceptable data loss for data that is being read into Advanced Revelation. 0=no data loss allowed; 1=some data loss allowed; 2=total data loss is allowed. When the specification is 1 or 2, there may be additional overflow and underflow values specified. The overflow value is substituted whenever a number is too large to read in. The underflow value is substituted whenever a number is too small. For more information, see "Introduction to Bonding" in <i>Environmental Bonding</i> .

Position 40 Output	For bonded files, the Output specification sets the degree of acceptable data loss for data that is being written to a foreign environment. 0=no data loss allowed; 1=some data loss allowed; 2=total data loss is allowed. When the specification is 1 or 2, there may be additional overflow and underflow values specified. The overflow value is substituted whenever a number is too large to write out. The underflow value is substituted whenever a number is too small. For more information, see "Introduction to Bonding" in <i>Environmental Bonding</i> .
Positions 41 through 45 User Defined	Positions 41 through 45 are reserved for use by the developer. You may use these positions for any purpose that you like.
Position 46 through 50 Reserved	These positions are reserved for use by Advanced Revelation.
Position 51 and Beyond Object Code	Position 51 stores the compiled dictionary object code. This position and all of the remaining positions in the record are reserved for use by the Advanced Revelation dictionary compiler.



Chapter E 4

Advanced Revelation Volumes

An Advanced Revelation volume allows you to create or maintain files in any location, using any file type. This chapter explains what a volume is, how to access a volume, and how to create one.

About Volumes	E 4 . 1
Attaching Files in a Volume	E 4 . 2
Creating a Volume	E 4 . 4
Additional Volume Functions	E 4 . 5



About Volumes

While working with Advanced Revelation files, you will frequently run across references to “volumes”. The information in this chapter will explain what volumes are, how to access them, and how to create them.

Distributing Files

A very useful feature of Advanced Revelation is that you can create and maintain your files on almost any drive that you can get access to from the operating system. This can include local files, network files, and even files in remote locations such as on a mainframe or minicomputer. Basically speaking, if you can get access to a location from your PC (and have appropriate security rights), Advanced Revelation can store or retrieve data there.

A good practice is to keep all the files for any application separate from the files for another application. This makes it much easier to back up the data for an application. It also simplifies transferring data files to another computer if you are developing an application for someone else.

Different File Types

In addition to keeping your files in different locations, you can use different *types* of files. Advanced Revelation allows you to store your data in a variety of formats (in different file types), such as Advanced Revelation files, ASCII files, dBASE files, and others.

For an introduction to file types, see the chapter “Advanced Revelation Files and Records” in this section.

Specifying Location and Type

When you create a file, and later when you access it, you must tell Advanced Revelation both the location and file type of the file. For example, to create a file using the dBASE bond, you must tell Advanced Revelation both the location (usually the drive or path name) of the file. In addition, of course, you must tell it to create a file in dBASE III format instead of Advanced Revelation, ASCII, or another format.

To indicate both a file type as well as a location, you use a “volume”. A volume is a name that you assign to a group of files sharing the same file type and location.

For example, all the Advanced Revelation Linear Hash files on the path C:\AREV\PAYROLL would belong to a single volume, and all the dBASE files on the path D:\DBASE\AREVDATA would belong to a different volume.

If there were also Advanced Revelation Linear Hash files on the path D:\DBASE\DATA, they would appear on a different volume, because they are of a different file type.

To access the files just described, you would create different volumes. In the first case (Linear Hash files on C:\AREV\PAYROLL), you might assign the volume the name AREV_PAYROLL. For the dBASE files in the second example, you might use a name such as DBASE_DATA.

Attaching Files in a Volume

Before you access any file in Advanced Revelation, you must first “attach” the volume it is in. When you attach a volume, Advanced Revelation stores information about the file type and location of the files in the volume. From then on (until you log off), anytime you reference a file from that volume, Advanced Revelation “knows” its file type and location.

For example, imagine that you have a file called EMPLOYEES in the AREV_PAYROLL volume described above. When you attach the volume, Advanced Revelation will know that the EMPLOYEES file is an Advanced Revelation Linear Hash file, and that it resides on the path C:\AREV\PAYROLL. If you execute a command such as LIST EMPLOYEES, you do not have to indicate a location or file type for EMPLOYEES, because Advanced Revelation already knows that, based on the file’s volume.

When you attach a volume, all the files on that volume are accessible to you. If you prefer, you can attach specific files. In that case, the files that you have not specified are not attached.

To attach a volume or individual files, follow these steps:

1. Choose the options **Tools-Files-Attach** from the Advanced Revelation Main menu. The Attach window will display.
2. At the *Volume* prompt, enter the name of the volume you want to access. Press [F2] to see a list of volumes already attached or [Shift-F2] for a list of all volumes. If you leave this prompt blank, Advanced Revelation will assume that you want to attach the default data volume (see below).
3. At the *File list* prompt, enter a list of files to be attached from that volume. Leave this prompt blank if you want to attach all files on the volume.

☛ You can also use the ATTACH command at TCL to access files in a volume. See the “TCL Command Reference” chapter in *TCL* for information about this command.

Default Volumes

There are two volumes in Advanced Revelation that you do not need to attach before using. One is for Advanced Revelation’s system files and the other is a default volume for your data files. These volumes are attached automatically when you first log on to Advanced Revelation.

**The System Volume
(REVBOOT)**

Advanced Revelation maintains its system files (commands, system windows, etc.) in a volume called REVBOOT. The definition of this volume is:

File Type	Advanced Revelation Linear Hash
Location	Default directory at logon time.

**The Default Data
Volume (DATAVOL)**

Advanced Revelation also uses a default volume for your data files. If you do not indicate a volume name for certain operations, the volume name DATAVOL is assumed. These operations are:

- Creating a new file (DEFINEFILE).
- Attaching a volume or file (ATTACH).
- Detaching a volume (DETACH).
- Listing the contents of a volume (LISTVOL).

The default data volume uses the Advanced Revelation Linear Hash file type. When you first install Advanced Revelation, files for the default data volume are located in the subdirectory called DATAVOL immediately beneath the subdirectory onto which you installed your system.

For information about changing the location and file type of the default data volume, see the topic “Changing the Default Data Volume” later in this chapter.

Detaching Volumes or Files

If you no longer want to access a volume or file, you can detach it. To detach a volume or file, follow these steps:

- 1) From the Advanced Revelation Main menu, choose the options **Tools-Files-Detach**. The Detach window will display.
- 2) At the *Volume* prompt, enter the name of the volume you want to detach. If you want to detach only individual files, skip this prompt.
- 3) At the *File(s)* prompt, enter a list of files to be detached. If you leave this prompt blank, all files on the volume you entered at the *Volume* prompt will be detached. If you did not enter a volume name, all the files on the default data volume will be detached.

☛ You can also use the TCL command DETACH. See the “TCL Command Reference” chapter in *TCL* for information about this command.

Creating a Volume

Before you create or access files in Advanced Revelation, you must create a volume to tell Advanced Revelation about the file type and location of the files.

You do not need to create a volume if you use one of the default data volumes. See “Default Volumes” above for information.

To create a volume, you first assign a unique name to your new volume. You then tell Advanced Revelation the location and file type for the new volume. Finally, if you will be accessing bonded files, you might also indicate the location of the “control” files such as dictionaries, indexes, and the volume directory. Follow these steps:

1. From the Advanced Revelation Main menu, choose the options **Tools-Files-Set Volume**. The SetVolume window will display.
2. At the *Volume Name* prompt, enter a name for the new volume. Use a name that will help you remember something about the volume (such as ASCII_DAT or NETWORK.AREV) You can use any name, as long as:
 - It is unique.
 - It is 50 characters or fewer.
 - It contains no punctuation but periods (.) and underscores (_).
3. At the *File Type* prompt, enter the name of the file type for the volume. Press [F2] to see a list of possible file types.
4. At the *Data Location* prompt, enter the path or drive name for the volume. If you are using a remote storage location such as a database server or a larger computer, enter the name by which that location is recognized from your operating system.
5. If you are *not* using an Environmental Bond, skip the *Control Files Location* prompt.

If you are using an Environmental Bond such as ASCII or dBASE, fill in the path or drive name where you want to store Advanced Revelation system control files (dictionaries and indexes) that the bond requires. By entering a path at this prompt, you can keep your Advanced Revelation control files separate from the data files in your bond.

✎ For more information about control files, see the chapter “Introduction to Bonding” in the section *Environmental Bonding*.

6. Press [F9] to save the volume information.

✎ You can also use the SETVOLUME command at TCL to create a new volume. See the “TCL Command Reference” chapter in *TCL* for information about this command.

Additional Volume Functions

In addition to facilities for accessing and creating volumes, Advanced Revelation provides other volume utilities. These allow you to display lists of available files and volumes, change the location or file type of the default data volume, and establish or label a volume directory.

Displaying Lists of Volumes and Files

Advanced Revelation includes reports to display information about volumes and files that are currently available.

Displaying Available Files To see a listing of all the files that are currently available, choose the options **Tools-Files-List Files** from the Advanced Revelation Main menu. You can also use the command LISTFILES from TCL.

Displaying Files on a Volume To see a list of all the files on a volume, choose the options **Tools-Files-List Vol** from the Advanced Revelation Main menu. This will display the ListVol window. At the *Volume* prompt, enter the name of the volume for which you want to see a file listing. Press [Shift-F2] at the prompt to see a list of all volumes. The volume does not have to be attached in order for you to list the files.

You can also use the command LISTVOL at TCL.

Displaying Available Volumes To see a list of all volumes that have been attached during the current session, choose the options **Tools-Files-List Volumes** from the Advanced Revelation Main menu. You can also use the command LISTVOLUMES at TCL.

Modifying the Data Volume

Advanced Revelation uses the default data volume if you do not indicate a volume when you create or list files in a volume. When you first install Advanced Revelation, the location of the default data volume is a subdirectory called DATAVOL below the current subdirectory. The file type for the volume is Advanced Revelation Linear Hash.

You can change the location or file type for the volume, or both. This allows you to use a different subdirectory or drive as the default location for your files. It also allows you to use a different file type (such as Advanced Revelation ROS or dBASE) as the default file type.

To change the default data volume, follow these steps:

1. Create a new volume following the steps listed under “Creating a Volume” earlier in this chapter. When you do, enter your choice for the location and file type that you wish to use as the default.
2. From the Advanced Revelation Main menu, choose the options **Management-Environment-General**. This displays an environment window.
3. At the prompt *Default data volume*, enter the name of the volume you have just created. Save your change with [F9].

The next time you log onto Advanced Revelation, it will automatically attach the volume you have entered as the new default. In addition, if you create a new file without indicating a volume, it will be on the new default volume.

- ✎ If you have files on the old default data volume, you will need to attach them manually to access them. Advanced Revelation will no longer automatically attach the files. To attach these files, follow the steps under “Attaching Files in a Volume” earlier in the chapter.

Establishing or Labelling the Volume Directory

When you create the first file on a new volume, Advanced Revelation automatically creates a volume directory. Advanced Revelation uses the volume directory to keep track of which files are available on that volume.

When the volume directory is established, it is assigned a unique label (usually consisting of the date and time). If you like, you can establish the directory manually or assign your own label to the volume.

- ✎ You cannot change the label of a volume that has indexed files on it.

If you assign a volume label yourself, you must ensure that the label is unique among all volumes in your system. This guarantees that processes such as indexing will work properly.

To establish or label a volume directory, follow these steps:

1. From the Advanced Revelation Main menu, choose the options **Tools-Files-Name Volume**. The Namevolume window will display.
2. At the *Volume* prompt, enter the name of the volume you are establishing or labeling.
3. At the *Name* prompt, enter a new or changed label for the volume. The label must be unique throughout your application. Press [F9] to save your name.

You can also use the TCL command NAMEVOL to establish or relabel your volume directories. See the chapter “TCL Command Reference” in *TCL* for more information on this command.

Chapter F 1

Building Applications with Advanced Revelation

This chapter provides an overview of how to create an application with Advanced Revelation. Subsequent chapters describe in detail each of the steps in the development process, plus additional features of Advanced Revelation.

Advanced Revelation simplifies the process of building an application by giving you a complete set of fully integrated design tools. Using Advanced Revelation menus and commands, you can quickly build prototypes of applications and enhance them as required.

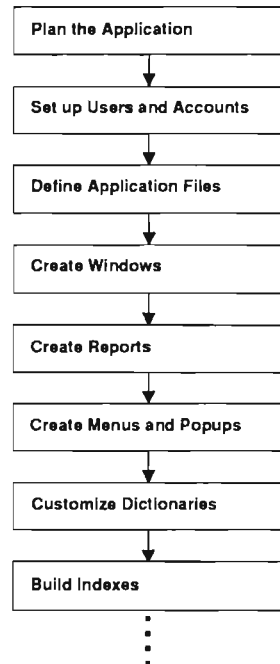
If you are not familiar with the principles of application design, review the “Planning an Application” chapter before you start building your application in Advanced Revelation.

Introducing the Application Development Process . . . F 1 . 1



Introducing the Application Development Process

Because of Advanced Revelation's flexibility, you can develop an application in many different ways. The general process, however, consists of these steps:



You will learn more about how to perform each of these steps as you read through the chapters in *Developing an Application*.

Essential Advanced Revelation Design Tools

Advanced Revelation offers many features that are not required for developing basic applications. Start by using the essential features. As you learn more about Advanced Revelation, incorporate more of the advanced features into your applications.

The following shows a typical outline of the Advanced Revelation features you will use at each stage of the development process:

When you want to:	Use:
Set up users and accounts	Management menu: all options
Define media, volumes, and files	File Tools menu: Name Media, Set Volume, Make File options
Design data entry windows	Design menu: Window option (Paint)
Design reports	Design menu: Report option
Design menus	Design menu: Menu option
Design popups	Design menu: Popup option
Customize dictionaries	File Tools menu: Dictionary option
Build indexes	File Tools menu: Indexing option

Some of the tasks listed above can be accomplished in more than one way. For example, in addition to using the Files menu to define indexes, you can also use special Paint options.

Limits Imposed by Environmental Bonds

Advanced Revelation’s design tools will accommodate a variety of file structures. However, specific environmental bonds may impose restrictions that are not accommodated by a particular tool.

If you are developing an application that uses environmental bonding, please refer to the *Environmental Bonding* section. The documentation for each specific bond provides details about any limitations that should be taken into account when working with Advanced Revelation tools.

Chapter F 2

Planning an Application

Careful planning is essential in creating an effective application. This chapter explains the steps required for planning a well-designed application, from assessing the user's requirements to designing the database files, windows, reports, and menus.

Planning Checklist	F 2 . 1
Establishing Application Requirements	F 2 . 1
Developing an Application Design	F 2 . 3
Designing the Advanced Revelation Structure	F 2 . 9
An Application Planning Worksheet	F 2 . 13



Planning Checklist

The following checklist gives an overview of how to plan and design an application. Each section in this chapter covers a different part of the process outlined on this checklist.

- Establish Application Requirements
 - ☐ assess application tasks
 - ☐ determine information needs
 - ☐ evaluate the user environment
- Develop an Application Design
 - ☐ group application information
 - ☐ detail information in each group
 - ☐ determine relationships between groups
 - ☐ determine account needs
 - ☐ refine the application design
- Design the Advanced Revelation Structure
 - ☐ organize database files
 - ☐ determine relationships between files
 - ☐ design application windows
 - ☐ structure security
 - ☐ design application reports
 - ☐ design application menus
 - ☐ design application popups

Each step in this checklist is described in more detail in the sections that follow.

Establishing Application Requirements

Before you start designing parts of an application, you need to assess the requirements of the application so that you can carefully plan a structure that will meet those needs. In this chapter, a computer store application will serve as a model for the planning process.

Assessing Application Tasks

When you design an application, you first need to examine the tasks the application must perform. You may find it helpful to interview the people who will be using the application, as well as their supervisors. This will give you a better understanding of the tasks their jobs involve. You may also want to talk with subject matter experts for technical details, such as accounting formulas or report format requirements.

For example, a computer store application might keep track of all the store's customers, the specifics of each sale, and the details of the products the store carries. An employee's daily tasks include writing customer invoices, calling customers, mailing orders, pricing equipment, ordering equipment, and tracking invoice payments.

Once you know what tasks the application must perform, you can organize tasks according to the order in which they will be performed.

If you start with a good understanding of what your application should accomplish, you can avoid the extra work of making last-minute additions and changes.

Determining Information Needs

A well designed application lets you extract the information you need from your database when you need it and in the form that is most appropriate. By looking at the paper records kept by a company or organization, you can get a good idea of the type and format of information required to run that operation.

Talk with managers to see how their current record keeping system works and how it might be improved in your application. A good design should address problems with the current system and provide alternatives. Give special consideration to any legal records that must be kept for tax or liability purposes.

In the computer store application, employees of the store will need to know the answers to a number of questions. Who are our customers? Who is the customer contact? What is the customer address and phone? What did each customer buy? When did they buy it? How much did it cost? What parts do we stock? What does each part cost? How many of each part do we have in stock? When did we last order each part?

To address these questions, the application must allow employees to:

- Enter and look up contact names and phone numbers for customers
- Print lists of customer phone numbers
- Print customer mailing labels
- Enter and look up the details of each sale
- Print detailed invoice slips for each sale
- Look up invoice amounts for each customer
- Enter and look up inventory, order, and price details for each product

For your own applications, create a list of required information like the one shown above. This will be useful when you begin building the application in Advanced Revelation.

Evaluating the User Environment

The environment in which an application is to be used can affect the design of an application. How many different users will work on the system?

If you will be creating a system for a network, you might need to be more concerned with security, and with making sure that users do not interfere with one another as they work. Design your application taking into account the system resources available (printers, hard disks, etc.), and the growth of the network.

It is valuable to create a list of all of the system users and the kind of information each user will need. For example, the computer store application might have the following users:

Chart of User Requirements for Proposed System

User:	Needs to:
Salesperson	issue invoices, check prices, call customers
Manager	check sales figures, oversee product orders
Data entry person	enter customer addresses, enter product information
Inventory clerk	track products in stock, order new products

Such a list will help you later as you set up security restrictions for the system. For more information on setting up users and security, see the “Users and Accounts: Planning Access and Security” chapter.

If your application will have many different users who will need to perform different tasks, you might consider breaking the application up into several different applications. For example, an accounting application might be split into an Accounts Receivable application, an Accounts Payable application, and a General Ledger application.

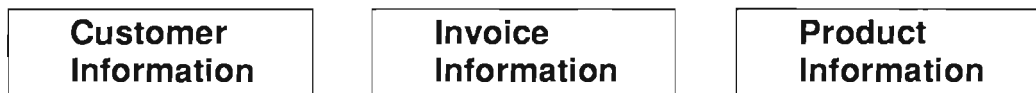
Developing an Application Design

Once you know what the application must accomplish and how it will be used, you can design your application to meet those requirements.

Grouping Application Information

Organize application information into logical groups based on related topics. But remember that Advanced Revelation lets you share information between groups, so you are allowed some overlap between your divisions.

In the computer store application, the information can be logically divided into three distinct groups:



You may decide later to subdivide your groups into smaller parts as required.

Detailing the Information in Each Group

List all the information each group will contain in as much detail as possible. Again, examine any available paper records to find out as much information as you can.

In the computer store application, each group contains its own unique set of information. Customer Information contains the company name, number, address, and contact for all the computer store's clients:

Customer Information

- Customer Number
- Company Name
- Contact Address
- City
- State
- Zip
- Phone
- Notes

The Invoice group contains all the details included on the computer stores standard invoice form:

Invoice Information

- Invoice Number
- Invoice Date
- Customer Number
- Company Name
- Product Code
- Description
- Quantity
- Price
- Amount
- Invoice Total
- Comments

The Product group contains details for each product the computer store stocks, including the product code, a description, price, quantity, the vendor who sells the product, and the date and quantity of the last order:

Product Information

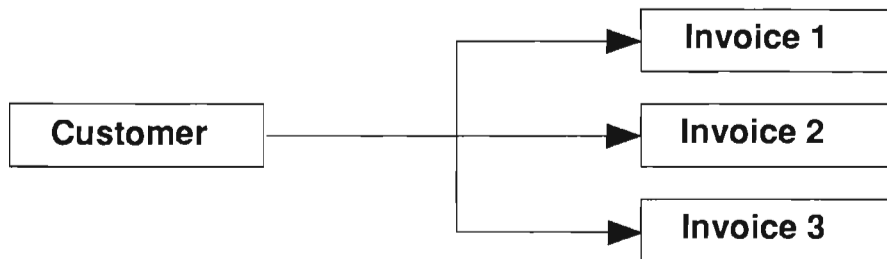
- Product Code
- Description
- Price
- Quantity on Hand
- Vendor
- Order Date
- Order Quantity

The more detail you can gather at this stage of the planning process, the easier it will be to design the Advanced Revelation structure later.

Determining the Relationships Between Groups

It is crucial that you understand how each information group will interact with other groups before you start designing your database. Determine the unique and shared information for each group and sketch the group relationships.

In the computer store application, each invoice is associated with a particular customer. A customer might have a number of invoices, but each invoice has only one customer.



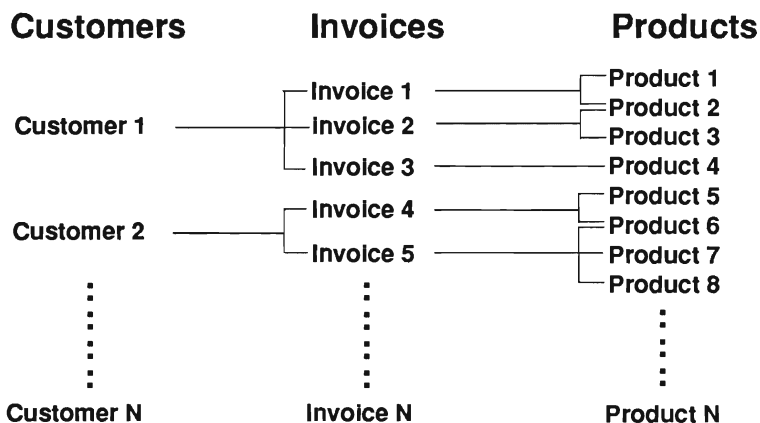
The relationship shown above is established through the number assigned to each customer. Each invoice references this customer number.

In the computer store application, some information is also shared between the invoice group and the product group, as shown below:

Invoice Group	Product Group
Product Code	Product Code
Description	Description
Price	Price
Invoice Number	Quantity on Hand
Invoice Date	Vendor
Customer Number	Order Date
Company Name	Order Quantity
Quantity	
Amount	
Invoice Total	
Comments	

The relationship between invoices and products is established using the product code.

Here is a visual overview of the group relationships for the computer store application:

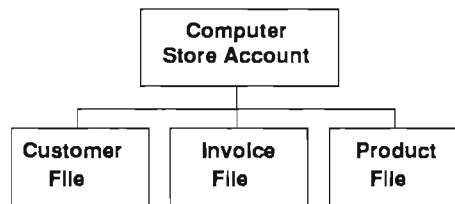


Relationships between different files in the database are generally established through relational indexing. For more information, see “Establishing Relationships Between Files” later in this chapter.

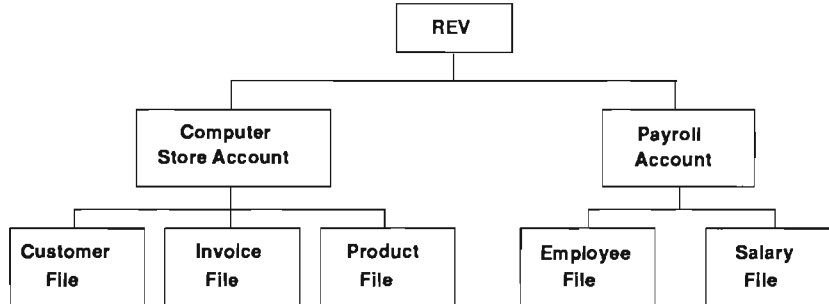
Determining Account Needs

It is generally a good idea for each application to reside in its own account. An account is simply a logical grouping of files. By setting up an application in its own account, you can restrict access to an entire group of files by establishing security for that account.

You could logically group the files for the computer store application into a single account:



If you were to create another application to run on the same system, such as a payroll application for the computer store personnel department, you might set up another account for Payroll:



Each department can have its own account, and information may or may not be shared between different accounts as needed. Use an account if each department has its own application that must be kept separate or if different users on the same system have discrete applications that must be kept separate.

- ☛ You must use an application account if you plan on using window security in your application. Window security is disabled in the system programmer (SYSPROG) account.

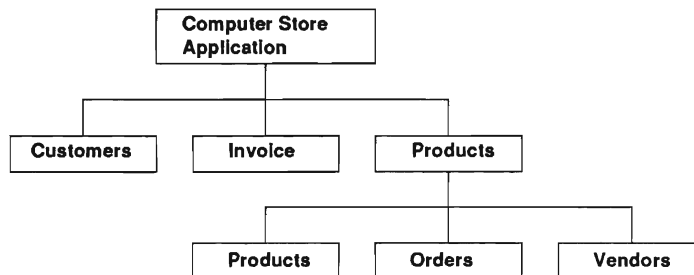
For more information on accounts and account security, see the "Users and Accounts: Planning Access and Security" chapter.

Refining the Application Design

Once you have a basic idea of the information your application will contain, look for ways to simplify tasks, reduce unnecessary repetition, and minimize mistakes.

Find places where mistakes are critical, and plan help messages, popups, edit checks, and security accordingly. Run through a sample scenario of using the application. Find any points where information is still missing.

In examining the computer store application, for example, you might decide that you need a way to track which equipment vendors respond the fastest to equipment orders. The application could be expanded by adding more product information and dividing the product group into three parts — Products, Orders, and Vendors:



Each of these new product groups could contain the following information:

Products

- Number
- Description
- Stock Price
- Alternate Products
- Alternate Price
- Order Price

Orders

- Vendor Number
- Order Date
- Order Quantity
- Average Response Time

Vendors

- Vendor
- Address
- City
- State
- Zip
- Phone

This new computer store application structure lets you track which vendors have the best order response time by making queries against the Orders and Vendors files.

For your own applications, you may go through several cycles of refinement. One of the advantages of using Advanced Revelation is that you can enhance and expand your applications as needed without rebuilding the application. You can create a basic prototype and then keep adding features at any stage of the application's life cycle until you are satisfied.

Designing the Advanced Revelation Structure

Once you have a basic application design, you can design the database files, windows, security, reports, popups, and menus for the application.

Organizing the Database Files

Assign groups of tasks to files based on topics. In general, each logical group will have its own file and each file will have a particular relationship with other files.

The computer store application contains a file for each of its three groups — Customers, Invoices, and Products:

Computer Store Database

Customer File	Invoice File	Product File
------------------	-----------------	-----------------

If you divided products into three separate groups — Products, Orders, and Vendors — you would make a separate file for each group.

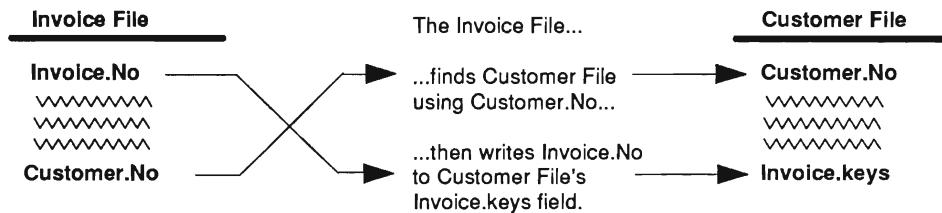
The information that each file will contain is determined by its data dictionary. For more information see the “Advanced Revelation Dictionaries” chapter and the “Managing Files and Records” chapter in the *Filing Systems* section.

Establishing Relationships Between Files

Once you have organized the files for each group, sketch out the relationships between files. Files are related through individual fields in each file.

Advanced Revelation files are linked together through a process called Relational indexing (joining one file to another). Files that have been related together allow you to access, display or update information from several different files at once.

In the computer store application, the customer and invoice files are linked together by the number assigned to each invoice and the number assigned to each customer. The invoice number field in the customer file keeps track of the invoice records for each customer. The customer number in the invoice file keeps track of the customer with which each invoice is associated.



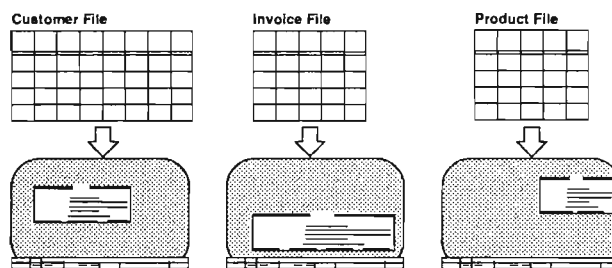
For more information on Relational indexing, see the "Creating and Maintaining Indexes" chapter.

Designing Application Windows

Windows provide a way for users to enter, edit and view data. Windows are like electronic versions of paper forms that users "fill out" with information required in that file. Usually, each file will have its own window.

You will probably want to sketch the layout of windows before you actually start building them. You should organize the order and position of information on the windows so that the flow of the window follows that of the process being performed.

The computer store application has three main windows, one for each of its files:



Each of these windows is organized in a format that lets you view database information quickly and easily.

Consider several techniques when designing your own windows, following these guidelines:

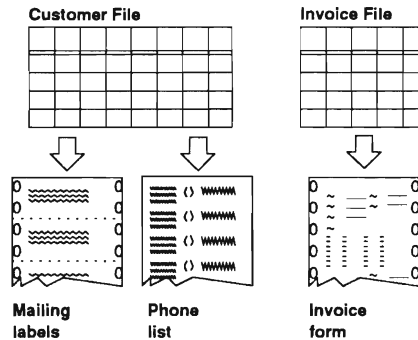
- Make sure you can easily distinguish between different fields in the window.
- Group fields together logically.
- Put the most frequently used fields at the top of the window, and the least frequently used fields at the bottom.
- Divide windows into several pages if it helps clarify the organization of prompts.
- Duplicate the layout of existing paper forms if they work well.

Designing Application Reports

Using the information you gathered at the beginning of the planning process, you can determine which reports are needed for your application. You might have weekly progress reports, annual financial reports, or timesheet report forms that are filled out and printed daily. Use any existing paper forms as a model for designing your own reports.

Advanced Revelation offers you four kinds of reports: forms, labels, lists, and reports that combine and format text with varying quantities of data. Forms are often used for information such as invoices, in which you want to arrange the layout of fields. Labels are used for printing mailing labels. Lists are used for general information reports when you want to see the contents of a file in columns and rows. Merge Reports are best suited to creating forms for variable data.

For example, the computer store application might use reports that produce invoice forms, customer mailing labels, a phone list, and sales letters personalized for each customer. Each of these reports would print information from a particular file. Invoices would be printed using data from the Invoices file, while the other reports might draw their data from the Customer file.



Designing Application Menus

Menus provide a convenient way to get in and out of application windows and to print reports. They tie the different parts of the application together and make it easy for users to move between parts. The way you organize your menus will be determined by the order and frequency of the tasks the application performs.

The menu in the computer store application offers the user five different options:

- Customer
- Invoice
- Product
- Reports
- Main Menu

You can create different menus for different users on the system. For example, you could create a menu that was a subset of the menu above, containing only the first, third, and fifth options. This menu might be displayed for a data entry person using the application, while the full menu remains available for a manager.

You usually want to design menus after you decide which windows and reports your application will use. It is a good idea to group functions together on your menus according to the order or frequency of use. For example, you might want to put all reports on a single menu, with the most frequently run reports at the front of the menu.

Designing Application Popups

A Popup is a window that “pops up” to offer a user some options to choose from, or to display text on the screen. Popups are often used with the [F2] Options key to show users what options they have when they are being asked to enter information. Popups can also be used to display help messages.

For example, your application may ask a user to enter a product code on an invoice. You could create a popup that displays all available product codes for the user, so that the user could simply pick the appropriate code from the list.

Examine your application and anticipate places where users may need additional information not provided in a window. For more information on popups, see the “Building Menus and Popups” chapter.

An Application Planning Worksheet

This section summarizes the steps in the application planning process. Use it as a worksheet for planning your own applications.

Establishing Application Requirements

- ☐ Define the application objective.
- ☐ Interview the people using the application, if you are computerizing an existing application.
- ☐ Interview subject matter experts.
- ☐ Determine the tasks an application must perform.
- ☐ Organize the tasks in order of use.
- ☐ Examine any existing paper records.
- ☐ Talk to managers about how their current record-keeping system works.
- ☐ Consider any legal requirements the application must meet.
- ☐ List all the questions a user may need to answer.
- ☐ Determine what information the user needs in order to answer each question.

Developing an Application Design

- ☐ Organize the application information into logical groups.
- ☐ Outline the information each group will contain.
- ☐ Determine the unique and shared information for each group.
- ☐ Sketch the group relationships.
- ☐ Determine if a separate account is needed for the application, or for discrete parts of the application.
- ☐ Refine the application design by simplifying tasks and reducing repetition.
- ☐ Determine where mistakes are critical and where help messages can minimize mistakes.
- ☐ Make sure the application design meets the requirements established earlier.

Designing the Advanced Revelation Structure

- ☐ Choose a file for each related group of tasks.
- ☐ Sketch the relationship between files.
- ☐ Sketch the structure of application windows.
- ☐ Design a security structure for the application.
- ☐ Determine which reports are needed for the application.
- ☐ Determine the menus and popups needed for the application.

Chapter F 3

Users & Accounts: Planning Access and Security

This chapter explains what accounts and users are and how they can be used to set up security restrictions, environment characteristics, and video attributes for your applications.

Introducing Users and Accounts	F 3 . 1
Planning for Security	F 3 . 4
Setting Up Accounts	F 3 . 9
Setting Up Users	F 3 . 11
Setting Up Environments	F 3 . 13
Setting Video Attributes	F 3 . 15
Assigning Environment and Video Characteristics . . .	F 3 . 17



Introducing Users and Accounts

An account is a logical group of files. A user provides a unique environment for working with accounts. This section introduces users and accounts and explains the relationship between them.

You can protect information stored by Advanced Revelation by setting up accounts for each department or application, and by allowing users to access only those accounts with which they need to work.

Departments may choose to use more than one account in order to restrict access to information within the department. For instance, the Payroll department may logically be set up in one Advanced Revelation account, while Accounting may choose to divide its information into three accounts: Accounts Receivable, Accounts Payable, and General Ledger.

Logging On

The term “logging on” refers to how you enter Advanced Revelation to start a session. The logon command line that you enter tells Advanced Revelation which account and environment attributes to make available to you. Your logon command may also include one or more options to invoke special Advanced Revelation processing modes.

You can log on under a user name, an account name, or both. The syntax for the command line that starts Advanced Revelation is as follows:

```
AREV {user} {account} {/options}
```

- If you type just **AREV**, you will enter the SYSPROG account with the default user name, USER.
 - If you specify a user name after **AREV**, you will be given access to the account associated with that user.
 - If you specify an account name after **AREV**, you will be given access to the specified account.
 - If you specify a user and an account name, you will be given access to that account as a specific user.
- ☛ Passwords may be required at log on. For more information, see “Using Passwords” later in this chapter.

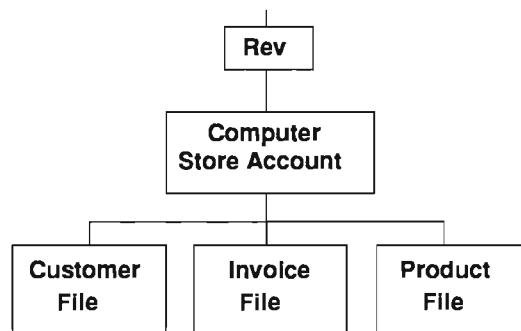
For more information about logon options, see “Logging On to Advanced Revelation” in the *Installation* section.

What is an Account?

An account is a collection of files that contains related information, such as payroll or inventory records. Accounts are used to group information together so that it can be easily located and protected.

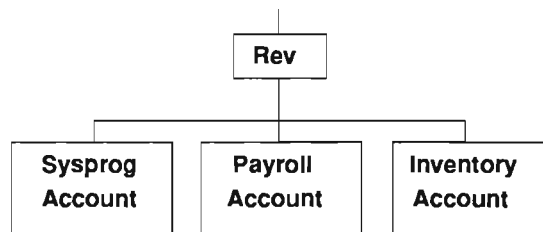
You can use accounts to protect sensitive information and to segregate different applications, which is particularly important in a networking environment.

The account you log onto determines which files are available to you. For instance, a computer store application might logically be organized into an account that contains these files.



When you create a new account, all system files needed to develop and run applications are made available to that account. You should then build your application in the new account.

An Advanced Revelation system might contain three different accounts as follows:



It is a good idea for each application to reside in its own account. For more information on creating accounts, see "Setting Up Accounts" and "Setting Up Users" later in this chapter.

Each account has its own set of security restrictions, environment characteristics, and video attributes. Security protects access to the account, and the video and environment settings determine how the application will look and behave.

Advanced Revelation does not limit the number of accounts on a system or the number of files in an account. Accounts can be added and deleted as needed.

What is a User?

A user is a set of attributes that defines how the Advanced Revelation environment will operate when you log on using that user name. By defining users for an account, you can make Advanced Revelation look and behave differently for each person using that account. This allows each user to work within the same account, using different environment settings. Each user record specifies a set of characteristics that determine access to information and the behavior of the application.

Users can have a unique set of environment characteristics and a set of video attributes, as well as an account and a restriction level.

By changing user environment settings, you can change characteristics such as how the Advanced Revelation text editor works or which menu will appear when a user logs on. For example, one user may have access to the TCL window, while another may see only the application windows and menus.

Video settings determine what application windows look like on the screen--if they have a reversed background, a highlighted border, color, and so on.

You can restrict the ability to log on for each user by specifying a password. You can also define a user restriction level to limit the user's access to secured application windows in an account other than SYSPROG.

The Relationship Between Users and Accounts

Both an account and a user can define the environment. Creating different user records for an account allows each user to use different environment settings while working within the same account. If no users are specified for an account, then everyone using that account will make use of the environment defined for that account.

Each user is associated with a single account. Most users work with one account, the one specific to their job. Users can access other accounts if they are given the account's password.

A user name carries its own environment and video specifications with it from account to account. When you log to another account from within Advanced Revelation, your user video and environment specifications are applied to the new account. The new account controls the logon verb, restriction level, and code and command specifications. Thus, your environment and video characteristics stay the same from account to account, while logon verb, restriction level, and code and command specifications may change based on the logged to account's definition.

The System Programmer (SYSPROG) Account

Advanced Revelation's System Programmer (SYSPROG) account is used to create all new accounts and users. When you first install Advanced Revelation and type **AREV**, you automatically enter the SYSPROG account.

All the menu options used for defining, securing, and deleting accounts and users are available only in SYSPROG. Once you have created a new account and defined users for that account in SYSPROG, you can log to that account with the Log To option on the Management menu. You should define all windows, reports, menus, popups, and files for your application in the account where they will reside.

For more information on creating new accounts and users, see "Setting Up Accounts" and "Setting Up Users" later in this chapter.

If you specify a password for the SYSPROG account, be sure to keep a safe record of it. If you forget the password, you will lose access to SYSPROG and will need to reinstall your system from an unprotected copy.

- ✎ Any window security specified for an application window is not checked when it operates in the SYSPROG account. However, an application window's security may be defined in SYSPROG and it will be operational when used in any another account.

Planning for Security

This section describes the types of security measures that are available in Advanced Revelation and how the security system works.

Using Passwords

A password is a set of characters that a user must type before being allowed to enter Advanced Revelation or display windows. You can assign a password to a user, an account, or a window. The system asks for a password only if a password has been assigned.

For example, you may assign a password to a particularly confidential account such as Payroll. Users can access that account only if they enter the correct password.

When you log on, you will be asked for a password for a user or account if one has been specified. You can optionally include a password after the user or account name on the command line using the following syntax:

```
AREV {user,password} {account,password}
```

- ✎ You must not include a space before or after the comma that separates the name from the password. If you do not specify a password on the command line but one has been defined for that user or account, the system will ask for one.

The following table shows the different ways passwords can restrict access when a user logs on to Advanced Revelation:

Table F3-1

How Passwords Can Restrict Access

Log on:	Asked for:	Gain access to:
AREV	No password	SYSPROG USER
AREV ACCT	Account password	Specified account
AREV USER	User password	Account assigned to user
AREV USER ACCT	User and account passwords	Specified account and user

A valid password produces the “Welcome to Advanced Revelation” message. Invalid passwords will result in error messages. After three invalid passwords are entered, the user is returned to the operating system.

Once you assign a password to a user, account, or window, the password is encrypted and cannot be viewed again. If you forget a password, you must create a new one.

Log to Passwords

You can set up several different passwords for users logging from one account to another within Advanced Revelation. These are known as “log to” passwords. Each log to password can have its own level of security, call its own unique logon verb, and have its own reset code and command. For more information, see “Securing Accounts” later in this chapter.

Using Multiple Passwords

Multiple passwords allow you to assign a second, temporary password to a user for temporary access to an account. When the user is finished working in that account, you can remove the second password, restricting that user’s access to that account.

For instance, you might allow data entry staff temporary access to a Payroll account to update information for tax records, and then restrict their access when the task is completed.

Assign multiple passwords to users and accounts in the same way you specify a regular password. For more information on how to set up passwords, see “Securing Accounts” and “Securing Users” later in this chapter.

Choosing User and Account Restriction Levels

Restriction levels restrict access to different windows in an account. You can assign each user and account a restriction level ranging from zero to five. Level zero (0) provides the least restriction and level five (5) provides the most.

Depending on how you log on, you will get a unique set of security restrictions and files for that particular Advanced Revelation session:

Table F3-2
How Logon Affects Security

Log on as:	Security Restrictions:
AREV	SYSPROG - no restrictions apply
User	User
account	Account
User and account	Account

When you use a window, the system checks to see if there is a restriction level specified for that window. If the user restriction level is greater than the restriction level of the window, then you are asked for a password. If the window does not have a password, you are denied access. For more information on window security, see “Restricting Access to Windows” below.

If you log to a new account, you may get a new set of security restrictions:

Table F3-3
How Logto Affects Security Restrictions

Log to:	Security Restrictions:
An account using a log to password	Log to password
An account using no log to password	Account

If you log on to the SYSPROG account, you can change security settings for all the users and accounts on the system.

Restricting Access to Accounts

Users can be restricted from accessing any account by an account password. By including only those files a user needs on an account, you protect access to other files that they do not need to see.

Each account can be protected with a password. For example, the payroll department can protect a database containing sensitive salary information by assigning a password; only persons knowing the password would have access to the files in that specific account.

Restricting Access to Windows

You can protect the information in your database by denying users access to selected application windows. A window can be secured by specifying a restriction level, a password, and a list of accounts that have access to the window. The restriction level in a window defines which users have access to it. A user with the same or fewer restrictions than the window has immediate access. A user with a greater restriction level than the window is prompted for a password or, if no password exists, is denied access to the window.

The window checks two things: the account where the user is, and the restriction level of the user. If the account appears on the list of accounts specified for that window, then no password is required.

There may be more than one password defined for the window. Matching any one of the passwords allows access to the window.

The following table shows the results of combining different types of window security. In this table, a Y in the appropriately labeled column indicates that the account(s), restriction level, and/or password(s) have been specified for that window.

Table F3-4

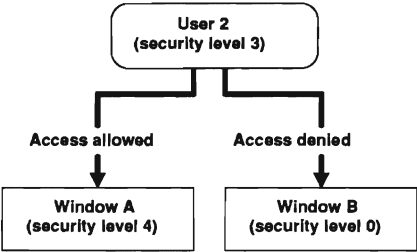
Planning Window Security

Accts	Restrict Level	Password	Result
N	N	N	Access is allowed.
Y	N	N	Access is allowed if account matches current account.
Y	Y	N	Access is allowed if account matches and window restriction is greater than or equal to current restriction level.

Table F3-4
Planning Window Security (Continued)

Accts	Restrict Level	Password	Result
Y	Y	Y	Access is allowed if account matches and current restriction level is less than or equal to the window's. If neither condition is met, access is allowed by typing the password.
N	Y	Y	Access is allowed if window restriction level is greater than or equal to current restriction level. If this condition is not met, access is allowed by typing a password.

For example, in the following illustration, the user has a security level of three. The two windows have security levels of four (low) and zero (high) respectively.



When user 2 tries to use window A, the system checks the user security level (3) against the window security level (4) and grants the user access. The user is denied access to window B, because of an inadequate restriction level. However, if there were a password specified for window B and the user typed in the password correctly, access would be granted.

The restriction level and account assigned to users depends on how they log on to the system. For more information, see “Choosing User and Account Restriction Levels” earlier in this chapter.

For more information on setting up window security levels, see “Establishing Window Security” in the “Adding Advanced Features to Windows with Paint” chapter.

Setting Up Accounts

This section describes how to create and delete accounts and how to set up account security restrictions. You perform these procedures using the Management menu options.

To create and remove accounts, you must be in the SYSPROG account. Use the Account Management Menu for all account operations.

Making New Accounts

The Advanced Revelation system files are available to all accounts. When you create a new account, Advanced Revelation does three things. It creates a new record for that account in SYSPROG's SYSTEM file. It creates new files to hold the account's application templates and the system information for each application (such as the LISTS and QFILES files). It also creates a VOC file for the account and copies VOC items from the SYSPROG account to the new file. Each new account requires approximately 50K of storage space on your computer's hard disk.

For more information on system files, see the "Managing Files and Records" chapter in the *Filing Systems* section.

You use the **Make Account** option, available on the Account Management Menu, to make a new account. At the Make Account window, enter the name of the new account and press [F9] (Save) to create the account.

Deleting Accounts

You can delete accounts only from within the SYSPROG account. To delete an account, log on to SYSPROG and use the **Delete Account** option on the Account Management Menu.

- ☛ Be absolutely sure you want to delete the account before giving this command. It permanently removes the account and all of its related files.

If you are on a network, make sure no one is logged onto that account before deleting it.

When you choose **Delete Account**, you are asked for the account name. Press [F2] (Options) for a list of all the accounts currently on the system. After you enter an account name, press [F9] (Save) to delete the account.

Securing Accounts

You can secure accounts only from within the SYSPROG account. Use the **Secure Account** option, available on the Account Management Menu. Security options include a restriction level and password. The Secure Account window appears as follows:

Secure Account					
Account					
Logon verb					
Restrictions					
Environment					
Video					
Password					
Restart	Code	Command			
Passwords for LOGTO Command					
Password	Logon verb	Restrictions	Restart	Code	Command
Window	Enter				Collector
Ed Off				Sysprog	Level 4

For more information, press [F1] (Context help) at any prompt in the window. When your entry is complete, press [F9] (Save) to save it. Descriptions of each prompt follow:

- Account** Specifies which account this set of security restrictions applies to. The account must first be defined with the Make Account window.
- Logon Verb** Specifies the name of a process that will be run when a user first logs onto the account. Logon verbs are commonly used to attach files. They are similar to an AUTOEXEC.BAT file in MS-DOS. You define a logon verb as a record in the VOC file using the Advanced Revelation Editor. This record can either be a catalog pointer or a TCL batch command.
- Restrictions** Indicates which restriction level to assign to this account. This level is compared with a window's restriction level to determine access to that window.
- Environment** Specifies a particular set of environment characteristics to assign to this account. For more information, see "Setting Up Environments" later in this chapter.
- Video** Specifies a particular set of video attributes to display for this account. For more information, see "Setting Video Attributes" later in this chapter.

Password Indicates a series of characters a user must type to gain access to this account when logging on from the operating system. You can specify more than one password at subsequent entry lines at this multivalued prompt.

Restart Allows you to specify a code and command sequence to perform each time the system goes through its normal restart function. Restart codes and commands are executed immediately after the logon verb is executed. Restart also occurs when users type END from the Advanced Revelation debugger or when they explicitly enter the RESET command.

Restart codes and commands are commonly used for returning the user to a particular menu after the system is reset. This prevents users from gaining access to commands and menus you may want restricted. It also provides protection against users accidentally getting themselves outside of application menus and into Advanced Revelation system menus. For more information on codes and commands, see the "Using Codes and Commands" chapter.

Using Log to Passwords

When a user logs to another account from inside Advanced Revelation, they may be asked for a password. Depending on which password the user enters, they may get a different logon verb, restriction level, and restart code.

Log to passwords let you enforce security restrictions when a user logs to another account with the **Log To** option. Using the Secure Account window, you can specify multiple passwords that can each have their own restriction level and each execute their own logon verb and restart code and command. The prompts work in the same way as the standard account security prompts of the same name described previously in "Securing Accounts".

Setting Up Users

This section describes how to create and delete users and how to set up user security restrictions. You perform these procedures using the Management menu options.

You can create new users only from the SYSPROG account using the Users Management menu.

Advanced Revelation adds a new record to the SYSTEM file for each new user you create. For more information on system files, see the "Managing Files and Records" chapter in the *Filing Systems* section.

Making New Users

Choose the **Make User** option, available from the Users Management menu, to create a new user.

When the Make User window displays, enter the name of the new user. Save the user record by pressing [F9] (Save).

Once you have created a new user, you can set specific security restrictions for that user with the **Secure User** menu option, as described below.

Deleting Users

To delete a user, select the **Delete User** option from the Users Management Menu, specify the name of the user to delete, and press [F9] (Save) to carry out the delete operation.

☛ When you execute the Delete User command, the user record is permanently removed.

Securing Users

To secure a user, select the **Secure User** option from the Users Management Menu to display the following Secure User window. The prompts at this window are like some of the prompts available on the Secure Account window, except that they apply to a user instead of an account.

Secure User

User name

Account

Logon verb

Restrictions

Environment

Video

Password

Restart

Code Command

Window	Enter	Su	Options			Collector	
Ed Off				Sysprog		Level	4

Press [F1] (Context help) from any prompt at this window for more information. Also see the descriptions of account security prompts in “Setting Up Accounts” earlier in this chapter. Press [F9] (Save) to save your entry when you are finished.

Changing User Passwords

Users can change their own passwords by choosing the **Password** option from the Management menu. They select the option, type the new password, and press [F9] (Save) to save their entry.

Setting Up Environments

This section describes how to create new user and account environments using options on the Environment Management menu. To access this menu, choose the **Management-Environment** options from the Advanced Revelation Main menu.

What is an Environment?

An environment is a set of characteristics that affect the way an application looks and behaves. You can create a different environment for each user and account. An environment specifies these aspects of operation:

- TCL general system functions
- Help level
- Paint functions
- Indexing operations
- Editor functions
- Message display
- Query window functions
- R/LIST View window functions
- Macro key settings
- Keystroke capture
- Screen and printer display
- Multiuser functions (locking and data integrity)
- Concurrency control
- Serial port communications

Use options on the Environment Management menu to customize the standard environment settings provided in Advanced Revelation. Commonly changed environment settings include specifying whether to display the main menu or the TCL command window when first logging on to Advanced Revelation, setting tab stops, specifying a macro set, and turning Editor functions off and on.

You can also restrict access to different parts of Advanced Revelation through the **General** option on the Environment Management menu. Typical restrictions include turning off the TCL key, preventing macro set changes, and turning off keystroke capture.

Using the Environment Management Windows

You can change user and account settings within the Environment Management windows. These windows are accessed by using options on the Environment Management menu. To access this menu, choose the **Management-Environment** options from the Advanced Revelation Main menu.

Logging on to the SYSPROG account enables you to change any existing environment record. Individual users can change their own environment characteristics only if they are allowed access to the windows called by the Environment Management menu. You can restrict access to system windows by substituting your own set of menus.

To modify some aspect of the current environment, choose the relevant option from the Environment Management menu. For information on how to access and change different environments, see the next section, "Selecting an Environment to Edit".

For example, the General System Configuration window provides prompts used to specify what menu displays as the default system menu, the default system mode at logon, and the Help level (novice, intermediate, or advanced). To make changes to settings in the General System Configuration window, use the **General** option on the Environment Management menu.

Press [F1] (Context help) at any prompt for more information. If you make changes at the prompts in an Environment window, press [F9] (Save) to save the changes to the environment record.

- ☛ When you change an environment record, the new environment specifications are not in effect right away. They will be used the next time you (or the user whose environment you have edited) log into Advanced Revelation.

Selecting an Environment to Edit

When any of the Environment Management windows is displayed in the SYSPROG account, you can select different user environments to edit by pressing [Shift-F1]. This displays a popup that lists all the user and account environments on the system.

Choose the environment you want to edit from the popup list. The settings for that environment will appear in the current window, where you can make changes and save the new definitions by pressing [F9] (Save).

Creating a New Environment

Environment records are stored in the SYSTEM file on the SYSPROG account. Each environment record is assigned an identifying record key name that ends with the suffix “.ENVIRONMENT”. For example the default environment record that is supplied with Advanced Revelation has the key name DEFAULT.ENVIRONMENT.

When you want to define an environment that is different from the existing environment records, you can use the Record Copy window to copy an existing environment record to a new name. To create a new environment, log in to the SYSPROG account and copy the default environment record (DEFAULT.ENVIRONMENT) to a new record name (*name*.ENVIRONMENT). “*name*” is the environment name of your choice and “.ENVIRONMENT” is the suffix required for all environment records. You can then use the windows called from the Environment Management menu to access and modify the *name*.ENVIRONMENT record. To assign the *name*.ENVIRONMENT record to a specific user or account, use the Secure Account or Secure User window.

Setting Video Attributes

This section describes how to create new user and account video attributes using the Set Video window.

What are Video Attributes?

Video attributes affect how application windows, menus, popups, and messages look when they are displayed. You can create a standard set of video attributes for each user and account.

The Set Video window lets you customize the default video settings provided in Advanced Revelation. Video attributes include background highlighting, the characters used for window borders, and the selection of colors. Video attributes can be customized for each of the following:

- Account you logged onto
- User you logged onto
- Application windows
- Popup windows
- Menu windows
- Edit windows
- Message windows
- Help windows

Using the Set Video Window

User and account video settings are changed with the Set Video window, available from the Management menu. Video changes are made from the SYSPROG account, because users can change their environments only if they have access to the Management windows. You can restrict access to system menus by substituting your own set of menus for the user to work with.

You can display the Set Video window by choosing the **Set Video** option from the Management Menu. This window allows you to edit the set of video attributes for the account or user you are currently logged in under.

Use the [Pg Up], [Pg Dn], [↑], and [↓] keys to display all pages of the window. Additional pages of the Set Video window let you change the video settings for Menu, Edit, Message, and Help windows.

Selecting a Set of Video Attributes to Edit

You can select different video environments to edit from the SYSPROG account by pressing [Shift-F1] when the Set Video window is displayed. This calls a popup that lists all the video attribute records in the system. Choose the record you want to edit and press [Enter]. The settings for that video environment will appear in the Set Video window, where you can make changes and save the new definitions by pressing [F9] (Save).

Creating a New Set of Video Attributes

Video records are stored in the SYSTEM file on the SYSPROG account. Each video record is assigned an identifying record key name that ends with the suffix “.VIDEO”. For example the default color video record that is supplied with Advanced Revelation has the key name COLOR.USER.VIDEO.

When you want to define a set of video attributes that is different from the existing video records, you can use the Record Copy window to copy an existing video record to a new name. To create a new video record, log in to the SYSPROG account and copy the default video record (either DEFAULT.VIDEO or COLOR.USER.VIDEO) to a new record name (*name*.VIDEO). “*name*” is the video record name of your choice and “.VIDEO” is the suffix required for all video records. You can then use the Set Video window to access and modify the *name*.VIDEO record. To assign the *name*.VIDEO record to a specific user or account, use the Secure Account or Secure User window.

Assigning Environment and Video Characteristics

Depending on how you log on, you will get a unique set of environment and video characteristics for that particular Advanced Revelation session.

The following table shows the different environment and video settings Advanced Revelation uses when you log on.

Table F3-5

How Logon Affects Environment and Video Characteristics

If you log on under:	Advanced Revelation uses:
A user name	Environment and video settings for that user
An account name	Default environment and video settings for that account
A user name and account name	User's environment and video settings



Chapter F 4

Creating Windows with Paint

This chapter provides an introduction to the basic functions used to create any window. For further information on creating windows, see the “Adding Advanced Features to Windows with Paint” chapter. For details on creating report forms with Paint, see the “Creating Forms” chapter in *Creating Reports*.

Overview of the Painting Process	F 4 . 1
Getting Started	F 4 . 2
Adding, Changing, and Removing Prompts and Entry Lines	F 4 . 8
Adding, Changing, and Removing Labels	F 4 . 21
Using Paint Options	F 4 . 22
Resizing and Moving Your Window	F 4 . 26
Testing Your Window with Test Run	F 4 . 29



Overview of the Painting Process

Use Paint to create data entry windows and report forms. While using Paint to create a window, you can also define the file that will hold the window's data.

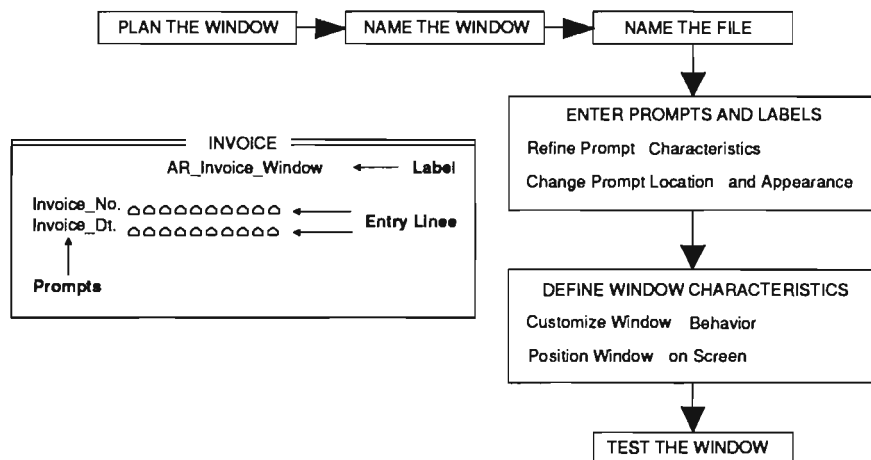
Windows in Advanced Revelation

In Advanced Revelation there are two kinds of windows: data entry windows and collector windows. Each serves a special purpose:

Data entry windows write data to or display data from a file. When you create a window for this purpose, you can either use an existing file to hold the data that the window processes, or you can name a new file and define the file's record structure at the same time that you create the window.

Collector windows collect information that will be used in a process.

Paint provides a set of tools used to define the appearance and function of a data entry or collector window. Using Paint, you create a window template record that is interpreted by the WINDOW command when the window is run in an application.



Paint allows you to add and edit prompts, labels and entry lines in a window.

- **Prompts** tell the user what information to enter. The characteristics specified for each prompt in a window define data verification requirements, default values, formatting, and user options and help. Prompts in data entry windows have an associated dictionary record that defines the characteristics of the field to which the prompt is related. Prompts in collector windows refer to a position in a variable.
- **Entry lines** are unique to each prompt. They are the area in the window where the user will enter data or view existing data when the window is run.

- **Labels** are used for titles, instructions, graphics, and general visual enhancements on the screen.

You can place these elements within the window and rearrange or modify the initial window layout as you like. You can also specify the size of the window, its position on the monitor, and any special video attributes.

Limits Imposed by Environmental Bonds

Advanced Revelation's application design tools will accommodate a variety of file types. However, specific environmental bonds may impose restrictions that are not accommodated by a particular tool.

If you are developing an application that uses environmental bonding, please refer to the *Environmental Bonding* section. The documentation for each bond provides details about any limitations that should be taken into account when working with Advanced Revelation's Paint tools.

Defining a New File's Structure with Paint

Paint is a prototyping tool that enables you to create both the dictionary and data portions of a file. As you design a data entry window with Paint, you can also define the record structure of the data file that will be updated by the window.

The dictionary portion of the file contains all of the information about the way the file should be structured. To define a new file, Paint begins by building the dictionary for the file. As new prompts are added to the window, records are added to the file dictionary. Paint creates a new dictionary record for each prompt field.

When you are satisfied that the window contains all of the prompts that you need to gather data for the file, Paint creates the data portion of the file. This may occur after one or multiple Paint sessions. Paint uses the field information stored in the dictionary to create the data portion of the file.

Getting Started

This section explains how to start Paint, how to save and delete a window, and how to leave Paint.

Starting Paint

To start Paint, choose the **Design-Window** options from the Advanced Revelation Main menu.

You can also start Paint by using the PAINT command at the Command Level (TCL). This is the general format for the PAINT command:

```
PAINT window.name
```

For more information on the PAINT command, see the PAINT entry in the “TCL Command Reference” chapter in *TCL*.

Naming the Window and File

When you start Paint, a window displays to prompt you for the name of the window. You can enter the name of an existing window so that you can make changes to the current definition. Or, you can enter a new window name in order to create a new window.

Modifying an Existing Window

If you want to modify an existing window, enter the window's name. To choose from a list of available windows, press [F2] (Options).

When you enter the name of an existing window, Paint displays the window definition so that you can change it.

Creating a New Data Entry Window

If you want to create a data entry window, enter the new window name. Your window name can contain up to 50 alpha-numeric characters, including dollar signs (\$) and underscores (_).

Paint centers the window name at the top of the screen. Then Paint displays another window, to prompt for a file name.

You can create a new data entry window that reads and writes to an existing data file or you can use Paint to design a new data file for your window to use.

Using an Existing Data File

To create a window that will access an existing data file, enter the name of the file. Press [F2] (Options) for a list of available files.

 You can use the **Quick Paint** option to create a window for an existing file. For more information, see the chapter “Adding Advanced Features to Windows with Paint” in *Developing an Application*.

Creating a New File

To create a new file, enter the new file name. The filename may contain up to 50 alphanumeric characters, including dashes, dollar signs and underscores. Do not use periods or parentheses in the filename and do not begin the file name with a number. The filename ends at the first occurrence of a space.

For compatibility with SQL, we recommend that you follow these conventions when specifying a filename:

- use up to 18 characters
- begin with an uppercase alphabetic character

- use no punctuation except underscore (_)
- use only alpha and numeric characters (plus underscore)

After you fill in this window, press [F9] (Save) to save your entry.

If you are creating a new file, Paint displays the Make Dictionary window.

Your entries at the Make Dictionary window tell Advanced Revelation where to put both the data and dictionary portions of the new file when they are created. Help is available at any prompt in the Make Dictionary window. Just press [F1] (Context help) or [Ctrl-F2] (Concept help).

The new dictionary name appears at the *Dictionary name* prompt. To indicate where the dictionary and file should be located and the file type you want to create, enter a volume name at the *Volume* prompt. For a list of attached volumes, press [F2] (Options). If you do not make an entry at the *Volume* prompt, the dictionary and file will be created on the default data volume DATAVOL as an Advanced Revelation file.

- ☛ Your environment may specify a volume other than DATAVOL as the default data volume. The default data volume is specified at the *Default Data Volume* prompt in the General System Configuration window. To access the General System Configuration window, choose **Management-Environment-General** from the Advanced Revelation Main menu.

Depending on the file type that you are creating, you may want to specify attributes that apply to this file. To do so, press [Shift F1]. Paint will display a collector window that prompts for file attributes. Press [F1] (Help) at any prompt in the window for help making a specification. Fill in the prompts and press [F9] to save your specifications.

- ☛ For information about the file attributes used with Advanced Revelation files, see the “Managing Files and Records” chapter in *Filing Systems*. If you are creating a bonded file, see your bond documentation for an explanation of the file attributes you can specify for that file type.

After filling in the Make Dictionary window, press [F9] (Save) to save your entry and create the dictionary portion of the file.

Creating a New Collector Window

If you want to create a new collector window, enter the new window name.

Paint centers the window name at the top of the screen. Then Paint displays another window, to prompt for a file name.

Collector window do not access data files, so you do not enter a data file name. Instead enter the word:

Collector

- ☞ For more information about defining collector windows, see the chapter “Adding Advanced Features to Window with Paint” in *Developing an Application*.

Virtual Space

Once you have specified the window name and created the dictionary file, Paint displays the window so that you can work on it. Except for the status lines at the bottom, a new window fills the entire screen.

You are not limited to the size of the screen in creating your window. In fact, the maximum area you may use is 32,000 bytes. If you used the maximum area in a square, that translates into a layout 180 columns wide and 180 rows deep. This is the complete area you use for your window's prompts, entry lines, and labels.

- ☞ This is a theoretical limit. The actual limit may be determined by the amount of memory available.

When you create your window, you position the prompts and labels where you want them to appear on the screen. Then you can resize the window's borders to display only part of your window's virtual space at one time.

Saving a Window

To use the new window for data entry, the window template must be saved. At the end of a window painting session, press [F9] (Save) to save all additions and changes to a window template record. Advanced Revelation writes the new or updated window template record to the TEMPLATES file. Your window must contain at least one prompt and entry line to be written to the TEMPLATES file.

Deleting a Window

If you have no use for a defined data entry window, that window template record can be deleted from the TEMPLATES file. Paint provides two ways to delete a template record.

- ☞ If you performed a test run on the template, deleting the template will also delete the temporary test run file created for the test.

To delete a window template from the TEMPLATES file, start Paint and enter the name of the window you want to delete. When the window appears, press [F10]

(Menu) to display the Paint menu. Choose **More** to display the second Paint menu. Then choose **Delete**. Paint displays a message informing you that your current window is about to be deleted from the TEMPLATES file.

To delete the window, press Y. To halt the delete process and keep the window, press [Esc] or [Enter].

Paint also accepts the standard “delete item” keystroke [Alt-D]. This enables you to delete a template while within Paint without using the **Delete** option from the More menu in Paint.

To delete the current window, press [Alt-D] while in Roam mode. Paint will confirm that you wish to delete the template by asking you to repeat [Alt-D]. Once you have done so, the template will be deleted.

- ☛ Paint always maintains a backup copy of a window in the TEMPLATES file. This backup copy is not deleted when using [Alt-D] or the **Delete** option from the Paint menu. To delete this backup copy, use the DELETETREC command or window. A backup copy is identified by the characters “.BAK” at the end (for example, the backup version of CUSTOMER_ENTRY is called CUSTOMER_ENTRY.BAK).

Leaving Paint

To save any changes that you made to the window template, press [F9] (Save) before leaving Paint. To leave Paint, press [Esc]. Paint asks you to confirm that you want to exit. Press [Esc] to confirm or press any other key to remain in Paint. Depending on circumstances, when you press [Esc] to confirm, Paint may ask a series of questions. Your responses direct Paint to perform various functions described below:

Creating the Data File

If you are creating a data entry window, and the data portion of the file does not exist yet, Paint will ask if you want to create the data file at this time. If you are not ready to create the data file at this time, enter N. To create the data file, enter Y.

- ☛ It is a good idea to refrain from creating the data file until you are completely done designing the window. Once the data file is created, a bonded environment may not allow you to alter the structure of the file without deleting it first. For more information, see your bond documentation.

Saving Your Test Data

The Test Run option in Paint allows you to test your window and save records to a temporary test run file. If the window data file exists, and if you created test data by running Test Run, Paint will ask if you want to copy your test run data to your real data file. To discard the test data, enter N. To copy the test data, enter Y.

Interpreting the Status Line

When you enter Paint, the status line displays information about your current activity. The illustration below shows the information that the status line displays when your help level is set to 0 (Advanced).

Paint	Roam	Col 2/30	Append mode	Auto bind	CUSTOMER	F4-edit F9-save
Ed Off		Row 2/21		Sysprog		Level 3

Process *Paint* indicates that the window or form painter is active.

Action Indicates the current mode.

Roam The editor is off. Use the keypad keys to move the cursor.

Prompt The editor is building a prompt label.

Entry The editor is building a prompt entry line.

Label The editor is building a label.

Column *Col##/##* indicates the current cursor position over the total columns in the window.

Edit mode Indicates the current edit operation.

Append Typed information is appended as a prompt at the end of the prompt sequence.

Insert Typed information is appended as a prompt at the end of the prompt sequence and the Prompt Order popup is displayed.

Label Typed information is added as a label.

Binding Indicates the kind of prompt to dictionary binding active.

Auto The prompt information automatically creates a new dictionary record of the same name.

Verify The prompt information can create a new dictionary record or be associated with an existing record.

Off The link between the prompt and a dictionary record is assumed. If a dictionary record exists, it will be read, otherwise, the prompt will not be associated with a dictionary.

Number	This process assigns a position number to a prompt. This is the normal binding mode for collector windows which have no dictionary.
Name	Indicates the name of the data file accessed by the window.
Message	Indicates that [F4] (Edit) can be used to edit a prompt, entry line, or label and that [F9] (Save) is used to save the template.

Adding, Changing, and Removing Prompts and Entry Lines

To enter prompts and entry lines on your window, position the cursor and enter the text. Paint automatically creates entry lines for you. This section explains the basic process of creating and changing prompts and entry lines. Adding and editing labels on the window is explained in the next section.

Overview

The application windows you create with Paint allow you to view and update records in a data file. Each prompt in the window corresponds to a field in the file. The data in each field is displayed in the window at prompt entry lines. Use Paint to define the characteristics for the prompts in your windows.

Each data file associated with a data entry or display window has a corresponding dictionary file that specifies the file's record structure. The dictionary contains records that specify the characteristics for each field in the record structure. The dictionary tells Advanced Revelation what to do with the data that is stored in the prompt's field.

In most database software, you must first create the field definitions before you can produce entry screens or reports. In Advanced Revelation, you can define your dictionary at the same time you define your windows or report formats.

The relationship between creating the prompts in a window and defining the fields in the dictionary is determined by the binding mode to which Paint is set when the window template is created or edited. The binding method determines the way your window's prompts relate to the dictionary.

Using the Autobind Binding Method

The default binding method in Paint is Autobind. Use this binding method to build dictionary records automatically as you enter your prompts. This method is commonly used when you are defining a new file while you design your window.

In Autobind mode, when you enter a new prompt, the binding process creates a new dictionary record. The dictionary record defines the characteristics for the field associated with the prompt. Paint copies the current prompt specification to the

dictionary to provide the field definition. The field name is the text of the prompt label you entered. The fields are numbered consecutively in the order in which you enter the prompts.

Other binding processes are available. For an explanation of when and how to change your binding method, see “Using Different Binding Methods” in the “Adding Advanced Features to Windows with Paint” chapter.

For more detailed information on creating dictionaries in Paint, see “Displaying, Editing, and Updating Dictionaries” in the “Adding Advanced Features to Windows with Paint” chapter.

Creating a New Prompt

- When you first enter Paint, the default editing mode is Append Prompts. Each new prompt you enter will be appended to the end of your prompt list. To change editing modes, use the softkey [Shift-F1]. For more information on editing modes, see “Choosing an Editing Mode” later in this chapter.

To create a new prompt, position the cursor where you want the prompt label to start and enter the prompt label. When you start typing, the Editor turns on and the entire prompt is highlighted. Use the standard editing keys to edit the prompt label text.

When the prompt label is the way you want it, press [Enter].

Assigning Field Names for Bonded Files

In Autobind mode, the text that you enter as the prompt label becomes the field name in the dictionary. However, some foreign environments have strict naming conventions. If you are creating a bonded file and your prompt label will create an illegal field name, Paint will display a message. The message indicates that the bond has assigned an alternate *foreign* field name for use in the foreign environment. The foreign name enables you to reference the field from within the foreign environment.

Press [Enter] to accept the assigned foreign name or enter the foreign name you prefer. The foreign name that you specify must comply with the field naming conventions for that environment.

For example, suppose you use the prompt label “Company Name” in a window that will update a dBASE file. This creates a field in the dictionary with the field name “COMPANY_NAME”. However, legal dBASE field names have a maximum length of 10 characters. So the dBASE bond assigns the foreign name “COMPANY_NA” for use in the foreign environment, which you can edit. The result is that Advanced Revelation knows the field as “COMPANY_NAME” and dBASE knows the field as “COMPANY_NA”.

Creating the Key Prompt

The first prompt you enter should be for the record's key field. Paint displays a message for you to confirm that the prompt will be for the key field. Press Y or [Enter] to confirm; press N if a different field will serve as the record key.

- ☛ For an explanation of record key fields, see the "Files and Records" chapter in the *Filing Systems* section.

Some foreign databases do not support the idea of key fields; typically they locate records according to their absolute location in the file. For these file types, the Environmental Bond will create an implicit key field at the time that the dictionary file is created. For example, both the ASCII and dBASE bonds create the implicit key RECNO in the dictionary. The data type for the implicit key is always INTEGER.

If the bond has created an implicit key in the dictionary, the window key prompt that you create in Paint will always be a synonym for the implicit key field. The bond will require that the key prompt be an INTEGER data type.

- ☛ The documentation for each Environmental Bond explains specific bond characteristics, such as the use of implicit keys.

Environmental Bond Limitation

When working with bonded files, you are free to add prompts and update the dictionary until the data portion of the file is created. If the data portion of the file already exists, there may be restrictions on adding new fields. Most bonded file types will not accommodate structural changes to an existing data file.

If you *must* add fields to an existing data file, Paint has to delete the current data file and create a new one that conforms to your changes.

If you choose to delete a file to accommodate the addition of a new field, Paint prompts you to confirm that you want to delete the file. If you enter Y at the prompt, the existing data file is deleted, along with all the data in the file.

Warning! Do not initiate this process unless you are willing to lose your data permanently.

At the close of the Paint session, Paint will offer to create a new, empty data file for you. The new file will reflect the new record structure in the current dictionary.

Specifying a Data Type

After you enter the prompt label, Paint prompts for a data type. A data type specifies the kind of information that will be stored in the field. Each data type consists of a name that identifies the data type and a variety of attributes that determine the characteristics of the field.

- ✎ If you work primarily with Advanced Revelation file types, you can disable data type prompting. To do so, use the **Management-Environment-Paint** options on the Advanced Revelation Main menu to access the Paint Environment window. Specify N (no) at the *Require data types* prompt in the Paint Environment window to turn off data typing. Henceforth, Paint will only prompt for data types if you are creating a bonded file that requires data typing.

The Data Types popup lists all of the available data types. Choose a data type from the popup to tell the system what kind of data the field will accept. For example, the INTEGER type tells the system that you want to accept whole numbers at the prompt. If you want the prompt to accept decimal numbers using a fixed decimal place, select the DECIMAL type. Or, if the prompt should accept a varying number of characters, specify the VARCHAR type.

If you are creating a bonded file, the Data Types popup may include bond-specific data types that correspond to the data types used in the foreign environment. The bond-specific data types appear at the end of the popup. For example, the popup for a dBASE file type includes a DBASE_M data type that corresponds to a dBASE memo field and a DBASE_D data type that corresponds to a dBASE date format.

Select the appropriate data type from the popup. If the type you select requires additional parameters, a window will display. Fill in the prompt(s) in the window and press [F9] to save your specification.

After you make the data type specification, a new dictionary record is created for the field. The characteristics of the field correspond to the data type specification for the prompt.

For an in-depth explanation of data typing see the “Files and Records” chapter in the *Filing Systems* section. For information about data types in bonds, see the “Introduction to Bonding” chapter in *Environmental Bonding*.

Completing the Prompt Entry

Paint automatically creates an entry line beside the prompt. The default length of the entry line is determined by the data type specification. If no data type specification is made, the default length is ten characters. To accept the current line length, press [Enter].

Changing the Size of an Entry Line

Before you can change the size of an entry line, the cursor must be in the entry line and the Editor must be on. Use [F4] to toggle the Editor on and off.

Use the arrow keys to change the size of an entry line. [→] and [←] expand and contract the right edge of the entry line.

The height and width of the current window border determine the limits to the size of a prompt's entry line(s). The entry line(s) can be extended up to the current window border, but not beyond it.

The actual length of the entry line in no way limits the amount of data that can be stored in the field. Your changes to the length of the prompt entry line only determines the number of characters that can display at the entry line. With Advanced Revelation's variable length architecture, a field can always contain more characters than are actually displayed at the prompt entry line.

- ✎ If the current entry line length was determined by your data type specification, changing the length of the entry line *does not* update the data type specification.

Adding Prompt Entry Lines

To add prompt entry lines the cursor must be in the entry line and the Editor must be on. Use [F4] to toggle the Editor on and off.

Use the arrow keys to add entry lines. [↓] adds more lines; [↑] removes lines.

- ✎ Your file type may not support multivalues. In this case, Paint will display an error message if you try to update the dictionary from the prompt.

If you are adding entry lines to create a multivalued prompt, you may display the multivalues horizontally as well as vertically. For example, multiple contact names for a customer might be displayed side-by-side or they might be displayed in a column.

To create a horizontal multivalued prompt in Paint, begin by painting a new prompt normally. Then use the [+] and [-] keys to increase and decrease, respectively, the number of values to be displayed at once. This is equivalent to using the [↓] and [↑] keys when creating vertical multivalued prompts.

As you press the [+] key, Paint will split the current display length of the prompt into multiple prompt entry lines, putting one blank column after each entry line. For example, if you begin with a display area of 10 columns and then press [+], Paint will split the prompt entry into two prompts of four characters each, with one column following each prompt display line.

As with vertical multivalues, the [→] and [←] keys increase the size of the individual prompt entries. All prompt entries increase or decrease at the same time.

The status line will report the current size and depth of your prompt. *Length* will indicate the total length of your prompt (not the individual length of each value). *Depth* will note how many horizontal values you have created.

For more information about multivalues, see the “Files and Records” chapter in the *Filing Systems* section.

Moving in Roam Mode

Two keystrokes will enable you to move around a window more quickly while in Roam mode.

The [Ctrl-Home] key moves you immediately to the top left corner of virtual space. The [Ctrl-End] key moves you immediately to the bottom right corner of virtual space.

Moving to an Existing Prompt

You can access any prompt by using the cursor keys to move the cursor onto the prompt or entry line for that prompt. To edit the prompt, toggle the Editor on by using [F4] (Edit).

With the Editor off, you can use Roam options to go directly to any prompt in your window. Press [F2] (Options) in Roam mode and choose the first option, **Find a prompt**. Paint displays a popup that shows you each prompt label with its dictionary name and dictionary position. Select the prompt to which you want to move the cursor. The Editor will be on for normal editing of that prompt.

Defining Prompt Characteristics

Each prompt in your window has a set of characteristics that tells Advanced Revelation what to do with the data you enter and display at the prompt entry line. When you create a new prompt, Paint gives the prompt a set of default characteristics. You can display and modify these characteristics at any time, using the Prompt window provided by Paint.

Use either of these methods to call the Prompt window:

- Move the cursor inside the prompt or its entry line and press softkey [Shift-F6].
- With the cursor inside the prompt or its entry line and the Editor on, press [F2] (Options) to display the Prompt Options popup. Choose **Prompt window** from the popup.

Using the Prompt Window

A prompt's current characteristics are displayed in the Prompt window. The characteristics can be modified by making entries at appropriate prompts in the Prompt window. At most of the prompts on this window you can press [F2] (Options) to see a list of entry options. Use the popup to make the appropriate entry at that prompt. For detailed help at any prompt, press [F1] (Detail Help) for information about what can be entered at that prompt. When your specifications for the prompt are complete, press [F9] (Save) to save the prompt definition.

Information entered in the first eight prompts in the Prompt window can be directly tied to the data dictionary through the binding process. When you use Autobind or Verify bind mode, the information in the Prompt window is used to create a dictionary record.

Specifically, the prompt entries used to create a dictionary record are:

- Field name
- Dict type
- Data type
- Position
- Display length
- Justification
- Validation pattern
- Output format

The specifications that determine a prompt's basic characteristics are described here. For an explanation of the other prompts in the Prompt window, see the "Adding Advanced Features to Windows with Paint" chapter. The relationship between the prompts in a window and the fields in a dictionary file is explained in that chapter as well.

Identifying the Prompt's Field

Three prompts in the Prompt window are used by the data dictionary to identify the field that holds the prompt's data.

Field name	Identifies the prompt's data dictionary field name. When you create a new prompt in Autobind mode, Advanced Revelation uses the prompt label as the fieldname. However, a prompt
-------------------	--

and its related field do not necessarily have to have the same name.

Dict type Shows how the prompt identifies the data for the field. These are the dictionary types:

Table number F4-1

Dictionary Types

Type	Description
F	The prompt points to a specific field in the record. This is the default field type.
S	Symbolic field. The prompt uses an R/BASIC formula to derive the data. Use a symbolic field to do calculations or read information from another field.

For more information on using dictionary type S, see "Using Prompt Formulas" in the "Adding Advanced Features to Windows with Paint" chapter.

Position Points to the position in the record where the data is stored. The position 0 (zero) is always reserved for the key to the record.

As you enter new prompts, Advanced Revelation assigns field positions to the prompts in sequence.

Changing the Field Data Type

The *Data Type* specification automatically updates four prompts in the Prompt window: *Display length*, *Justification*, *Validation pattern*, and *Output format*. Collectively, the specifications associated with the data type determine what data is accepted at the prompt and how it is stored and displayed.

Data Type Specifies the type of information that will be stored in the field associated with this prompt. Data typing can be used on any F or S type field.

You may assign a new data type to this field or change the current specification. Type in the correct data type or press [F2] to call the Data Types popup and make a selection. When you enter a data type, prompts in the Prompt window are updated to display the characteristics associated with that data type.

Environmental Bond Limitation

When working with bonded files, you are free to change the data type specification until the data portion of the file is created. If the data portion of the file already exists, there may be restrictions on changing the data type.

If you *must* change a data type specification after the data file is created, Paint has to delete the current data file and create a new one that conforms to your changes.

If you choose to delete a file to accommodate data type changes, Paint prompts you to confirm that you want to delete the file. If you enter Y at the prompt, the existing data file is deleted, along with all the data in the file.

Warning! Do not initiate this process unless you are willing to lose your data permanently.

At the close of the Paint session, Paint will offer to create a new, empty data file for you. The new file will reflect the new record structure in the current dictionary.

Determining How Data Is Displayed at the Prompt

The *Display length*, *Justification*, and *Output format* specifications define how to display data at the prompt. For an explanation of *Output format* see “Data Formatting” in the “Adding Advanced Features to Windows with Paint” chapter.

Display length

Shows the number of characters displayed at the prompt's entry line. Note that a field can contain more characters than are actually displayed.

To change the display length, return to the window definition and use the arrow keys to increase or decrease the size of the entry line. For detailed instructions, see “Changing the Size of an Entry Line” earlier in this chapter.

Justification Controls the justification for data displayed in the window or in reports. These are the format options:

L	Left justified
R	Right justified
T	Formatted text
TN	Unformatted text
C	Centered

Controlling Data Entry At a Prompt

The *Validation pattern* and *Entry type* prompts control the kind of data that Advanced Revelation will accept at the prompt. For an explanation of *Validation pattern* see “Data Validation” in the “Adding Advanced Features to Windows with Paint” chapter.

Entry Type Controls data entry at the prompt. Specifications made at this prompt allow you to protect data displayed at the prompt entry line. These are the entry types options:

Table Number F4-2

Prompt Entry Type Options

Type	Description
R	Required. The user must make an entry at this prompt before moving to the next prompt.
O	Optional. The user need not make an entry to move on.
P	Protected. The user may not make any changes to the data displayed at this prompt. The cursor will not move to this prompt, preventing edits.
F	Filled. When the record is created, the prompt's entry line is automatically filled with data from the Default prompt. For example, you might fill a date field with the current date. For information on setting the default value, see “Defining Advanced Prompt and Label Characteristics” in the chapter “Adding Advanced Feature to Windows with Paint.”
FP	Filled and protected. The prompt's entry line is filled, then protected. For example, you might fill in the prompt on a sales order entry window with today's date automatically, and prohibit the user from changing the date.

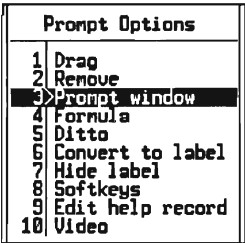
Table Number F4-2
Prompt Entry Type Options (Continued)

Type	Description
RP	Required and protected. The user must make an entry at this prompt. After leaving the prompt, the user cannot go back and change the data. This option is often used for the first prompt, or key field, in a record. It is also useful when the data in the field updates another file.
VP	Viewable and protected. Use this type to create a read-only prompt at any multivalued or text prompt. The user may not make any changes to the data displayed at the prompt. However, unlike the P (protected) entry type, the VP type allows the cursor to move to the prompt. This enables the user to scroll through the values at a multivalued prompt or through text at a text prompt where all the data may not be visible at once.

Displaying Prompt Options

Paint provides a popup that lists the available prompt options. To display and use these options, press [F2] (Options) when the cursor is positioned in the prompt and the Editor is on.

This is the Prompt Option popup:



Changing Prompt Color or Video Display

To change a prompt's color or video display when the window is used with a color monitor, go to the Prompt Options popup. To display the popup, place the cursor inside the prompt or its entry line, turn the Editor on, and press [F2] (Options). Choose **Video** from the popup to display the Select Video popup:

Select Video		
Video	Video	Literal
1 ON		Normal
2 R		Reverse
3 H		Highlight
4 B		Blink
5 RB		Reverse-blink
6 U		Underline
7 UB		Underline-blink
8 DBLUE		Dark-blue
9 BLUE		Light-blue
10 DCYAN		Dark-cyan
11 LCYAN		Light-cyan
12 DGRN		Dark-green
13 LGRN		Light-green
14 DYEL		Dark-yellow
15 LYEL		Light-yellow
16 DRD		Dark-red

pg 1/2

Choose the option you want.

Dragging Prompts to New Locations

Drag mode is used to move a prompt label, entry line, or set of prompts and entry lines to a new location in the window. With the cursor on the prompt or entry line you want to drag, use either of these methods to turn Drag on:

- Press softkey [Shift-F3]
- Choose **Drag** from the Prompt Options popup

Use the arrow keys to move the item. When you are finished moving it, press [Enter].

- ☛ When Drag is on, the status line displays information about the current location of the item you are moving and the size of the steps used to move it:
 - **Column** shows the prompt or entry line's present column position.
 - **by # Cols** shows the number of columns that the item will move at each press of the arrow key or step.
 - **Row** shows the prompt or entry line's present row position.
 - **by # Rows** shows the number of rows that the item will move at each press of the arrow key or step.
 - The Status Lines also tell you if you are moving a prompt, an entry line, both a prompt and its entry line, or a group of prompts together.

To drag a group of prompts, press [F2] (Options) when Drag is on. Choose the first option from the Drag Options popup, **Tag other prompts to drag as group**. Paint displays a popup listing all the prompts in the window. Choose the prompts to include in the group by highlighting each one and pressing [Enter]. If you change your mind about a chosen prompt, highlight it again and press [Enter]. When you have chosen each of the prompts you want to move, press [F9] (Save) to save your entries and return to your window. Use the [↓] [↑] [←] [→] keys to drag the group of prompts together.

To drag a prompt and its entry line together, press [F2] (Options) when Drag is on. Choose the second option from the Drag Options popup, **Drag prompt label and prompt entry together**.

Setting Step Size You can change the size of steps that the item moves at each press of an arrow key. The current step sizes are displayed on the status line.

Use these keys to change the size of the steps:

When you want to:	Press this key:
Increase row step by one	[Tab]
Decrease row step by one	[Shift-Tab]
Increase column step by one	[Spacebar]
Decrease column step by one	[Backspace]

Removing a Prompt

With the cursor on the prompt or entry line, use either of these methods to remove a prompt and its entry line from your window:

- Press [Shift-F4] two times.
- Choose **Remove** from the Prompt Options popup. Paint prompts you to press [Shift-F4] to confirm the removal.

Paint removes the prompt and its entry line, but does not delete the prompt's dictionary record.

☛ You may want to delete the associated dictionary record if the data file does not yet exist. If you do not, the field defined in the dictionary becomes part of the file structure, even though there is no prompt in the window. For information about removing fields from the dictionary, see the "Adding Advanced Features to Windows with Paint" chapter in *Developing an Application*.

Converting a Prompt to a Label

To convert a prompt to a label, choose **Convert to label** from the Prompt Options popup. Paint converts the prompt to a label and removes its entry line.

Adding, Changing, and Removing Labels

Use Label Edit mode to enter labels on your window. This section explains the basic process of creating and changing labels.

Turning On Label Edit Mode

You use the Label Editing mode to enter and change labels on your window. Press [Shift-F1] and choose **Label text** from the popup.

Entering a New Label

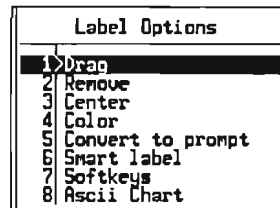
To enter a new label, position the cursor where you want the label to start and type the label. When you start typing, the Editor turns on and the entire label is highlighted. Use the standard editing keys to edit the label.

When you have typed the label as you want it, press [Enter].

Displaying Label Options

Paint provides a popup that lists the available label options. To display and use these options, press [F2] (Options) when the cursor is positioned in the label and the Editor is on.

This is the Label Options popup:



Centering a Label

You can center a label on its line in your window. This is convenient for centering titles or section headings.

To center a label, choose **Center** from the Label Options popup. Paint centers the label automatically for you between the currently defined left and right borders of the window.

Locating an Existing Label

Use Roam options to go directly to any label in your window. Press [F2] (Options) in Roam mode and choose the second option, **Find a label**. Paint displays a popup that shows you each label in the window. Choose the label you want. When you press [Enter] or [F9] (Save) to leave the popup, you go to the label and the Editor turns on.

Changing Label Color or Video Display

To change a label's color or video display, choose **Video** from the Label Options popup. Paint displays the Select Video popup, listing each video display and color option. Choose the option you want.

Dragging Labels to New Locations

Drag mode is used to move a label to a new location on the screen. With the cursor on the label you want to move, use either of these methods to turn Drag on:

- Press [Shift-F3].
- Choose **Drag** from the Label Options popup.

Use the arrow keys to move the item. When you are finished moving it, press [Enter].

For more information on Drag mode, see “Dragging Prompts to New Locations” earlier in this chapter.

Removing a Label

Use either of these methods to remove a label:

- Press [Shift-F4].
- Choose **Remove** from the Label Options popup.

The cursor must be on the label you want to remove.

Converting a Label to a Prompt

To convert a label to a prompt, choose **Convert to prompt** from the Label Options popup. Paint converts the label to a prompt and creates an entry line for it.

Using Paint Options

Paint provides a variety of tools that make designing data entry windows easier. This section introduces the options you will use to define your window.

Using the (F4) Editor Key

When you first enter Paint, the cursor displays as a block. This indicates that the Editor is off. As soon as you begin typing, the cursor changes to a line, indicating that the Editor is on. When your cursor rests in a prompt, entry line, or label with the Editor on, the item is highlighted and you can make changes to that item.

Some Paint options described in “Using the [F2] Options Key” are available only when the Editor is on. Use [F4] to toggle the Editor on and off.

Using the (F2) Options Key

Paint provides popups that list the options available to you in the current situation. The context in which a popup is called determines which options are listed. To display and use these options, press the Options key [F2].

These popups accomplish two things. First, by showing you the possible actions you can take, they help in your decision-making process as you create a window. Second, they take you to a specific task, or supply specific information relevant to your situation.

Displaying Roam Options

When you first enter Paint, the editor is off and you are in Roam mode. This means you can use the direction keys to roam freely around the screen and position the cursor.

Roam options help you quickly find any prompt or label in your window. To display and use Roam options, press [F2] (Options) when the Editor is off. This popup appears:



For details on the first two options, see “Locating an Existing Prompt” or “Locating an Existing Label” earlier in this chapter. Use the third option to display the Paint softkeys.

Using Softkeys

For the most part, Paint softkeys perform functions that affect your entire window. For your convenience, the options for dragging and removing a prompt or a label, and for displaying the Prompt window, are also included in Paint softkeys.

To display Paint softkeys, press [F6] (Softkeys). This popup appears:

Paint Softkeys		
	Key	Function
1	Shift F1	Modes
2	SF2	Bindings
3	SF3	Drag
4	SF4	Remove
5	SF5	Prompt Table
6	SF6	Prompt Window
7	SF7	Reorder
8	SF8	Default Window
9	SF9	List Dictionary
10	SF10	Edit Dictionary

You can also display Paint softkeys through the Roam Options popup, the Prompt Options popup, or the Label Options popup.

You can execute a softkey directly by pressing the appropriate keys. For example, the softkey [Shift-F3] activates drag mode. Either of these methods executes the softkey:

- Press [Shift-F3]
- Press [F6] (Softkeys) to display the softkey popup and choose **Drag** from the popup.

Choosing an Editing Mode

The editing mode determines whether you are creating prompts or labels.

- To create prompts, use **Append prompts** or **Insert prompts** as your editing mode.
- To create labels, use **Label** as your editing mode.

If you are creating prompts, choose Append prompts or Insert prompts depending on how you want to determine the data entry order of your prompts. While designing your window, you can create prompts in any order. However, when someone uses the window, you probably want the user to move from prompt to prompt in a specific order.

The default Editing mode is Append prompts.

Changing Your Editing Mode

To change your Editing mode, use softkey [Shift-F1]. This popup appears:

Select Mode	
1	Append prompts
2	Insert prompts
3	Label text

- Append Prompts** The data entry order of your prompts matches the order in which you create them. To change the data entry order, use softkey [Shift-F7] to display the Prompt Order Table. For more information on reordering prompts, see “Grouping and Reordering Prompts” in the “Adding Advanced Features to Windows with Paint” chapter.
- Insert Prompts** This mode is used when you add new prompts to an existing window. When a prompt is added, the order of data entry may not coincide with the prompt appearance order on the screen. After completing a new prompt definition, Paint automatically displays the Prompt Order Table. The new prompt is listed last in the data entry order. Use the Prompt Order Table to change the order of any prompt. For more information on reordering prompts, see “Grouping and Reordering Prompts” in the “Adding Advanced Features to Windows with Paint” chapter.
- Label Text** The characters you type will be used as labels, such as titles, instructions, or other text or graphics that will display permanently on the window. Labels are not associated with prompts.

Displaying a List of Active Keys

To display a list of active keys, press [Ctrl-F9]. Paint displays a list of keys with special actions, including function keys, cursor movement keys, and editing keys.

Displaying the Paint Main Menu

The Paint Main Menu provides options that relate to the entire window. To display the Paint Main Menu, press [F10] (Menu). Choose **More** from the first Paint menu to display the second Paint menu, More Paint Functions. Both menus are shown here:

Paint Main Menu			
Customize Relations	Groupings Softkeys	Processes Test Run	Quick Paint More
More Paint Functions			
Collector	Delete	Form Parameter Indexing	Key Disable
Recalculate	Security	Set Video Update	Overstrike
"Define 'Collector' window information."			

Resizing and Moving Your Window

This section explains how to use **Resize** and **Move** to change your window’s size and to position it on the screen. In addition, you will learn how to use **Pan** to move quickly through your window’s complete layout if its contents extend beyond its borders.

By resizing and moving your windows, you can design an application to display information from more than one window at once. For example, you might specify within **Paint** that a **Product** window is related to the **Invoice** window. Then you would size and position these windows so that neither occupies the same space on screen. When the **Invoice** window is run, the **Product** window can be called using an assigned softkey and you can see information from both windows at the same time.

For more information about relating windows see “Relating Windows” in the “Adding Advanced Features to Windows with **Paint**” chapter.

Changing Your Window’s Size

To change the defined size of the data entry window, press [Ctrl-F7] (**Resize**). Use the [↓] [↑] [←] [→] keys to push the bottom and right borders in the direction you want. You can use **Resize** to make a window as small as one column by two rows, or as large as the full screen. When you are finished resizing, press [Enter] or [Ctrl-F7] to turn **Resize** off.

- ☛ When **Resize** is on, the status lines display information about the current location of the window and the size of the steps used to resize:
 - **Column** shows the present column position of the window’s upper left corner over the total columns for the layout.
 - **by # Cols** shows the number of columns that the border will move at each press of the arrow key or step.
 - **Row** shows the present row position of the window’s upper left corner over the total rows for the layout.
 - **by # Rows** shows the number of rows that the border will move at each press of the arrow key or step.

Use these keys to resize the window’s right and bottom borders:

When you want to move the bottom border:	Press this key:
Up one step	[↑]
Up one half window depth	[PgUp]
Down one step	[↓]
Down one half window depth	[PgDn]

When you want to move the right border:	Press this key:
Right one step	[→]
Right one half window width	[Ctrl→]
Left one step	[←]
Left one half window width	[Ctrl←]

You can change the size of steps that the borders move at each press of an arrow key. The current step sizes are displayed on the status line.

Use these keys to change the size of the steps:

When you want to:	Press this key:
Increase row step by one	[Tab]
Decrease row step by one	[Shift-Tab]
Increase column step by one	[Spacebar]
Decrease column step by one	[Backspace]

Setting and Removing Page Tabs

For windows with layouts larger than the area displayed inside the border, you can define different sections as individual pages of prompts.

Mark the upper left corner of *each* page. When a person using the window presses [PgUp] or [PgDn], Advanced Revelation uses your markers to position the right set of prompts inside the window's borders.

[Alt-H] acts as a toggle to set and remove page tabs. To set a page tab that marks the top of a page in your window, follow these steps:

1. Position prompts for the first page as you want them to appear.
2. Position the cursor on the topmost prompt on the page.
3. If the Editor is off, press [F4] to turn the Editor on.
4. Press [Alt-H].
5. Position prompts where they should appear on the second page and repeat the page tab setting process.

Paint marks the top of the page with a special tab symbol Φ.

To remove a page tab, position the cursor on the tab symbol and press [Alt-H].

Moving Your Window

To move your window to a different location on the screen, press [Ctrl-F8]. Use the [↓] [↑] [←] [→] keys to move the window to the new location. The status line shows the current column and row position of the window's upper left corner.

When you are finished, press [Enter] to turn Move off.

Panning Through Your Window's Layout

To pan quickly through the layout of prompts when your layout is larger than the area displayed by your window's borders, press [F7]. Use the [↓] [↑] [←] [→] keys to pan through the layout. When you are finished, press [Enter] or [F7] to turn Pan off.

- ☛ When **Pan** is on, the status lines display information about what section of the window you are currently viewing and the size of steps used to pan through the layout.
 - **Column** shows the present column position of the window's upper left corner over the total columns for the layout.
 - **by # Cols** shows the number of columns that the layout will move at each press of the arrow key or step.
 - **Row** shows the present row position of the window's upper left corner over the total rows for the layout.
 - **by # Rows** shows the number of rows that the layout will move at each press of the arrow key or step.

Use these keys in Pan to pan through the entire layout:

When you want to Pan:	Press this key:
Up one step	[↑]
Up one page	[PgUp]
Down one step	[↓]
Down one page	[PgDn]
Right one step	[→]
Right one window width	[Ctrl→]
Left one step	[←]
Left one window width	[Ctrl←]
Home position	[Home]

You can change the size of steps that the layout moves at each press of an arrow key. The current step sizes are displayed on the status line.

Testing Your Window with Test Run

While defining your window, you can quickly test your design and then return directly to your window definition. This section explains how to use the **Test Run** option to test your window without ending the painting process.

Test Run is an option on the Paint Menu. Use **Test Run** to temporarily leave the painting process and put the current window into data entry mode. While in data entry mode, you can use either a test file or an existing data file to create new records, access existing data, test your window prompts, and generally get a feel for the current state of the window.

Performing a Test

To test run a window from Paint:

1. Press [F10] (Menu) to display the Paint Main menu.
2. Choose the **Test Run** option. Paint will save the window template in its current state and display your window.
3. If you are testing a window that accesses an existing data file, you can use the real data file for your test or you can test the window using a temporary test file. A popup displays so that you can make this choice. If the data file does not exist yet, the window test will automatically use the temporary test file.
4. Enter a variety of data to fully test the window.
 - If you decide to use the real data file during the test, any data that you save by pressing [F9] (Save) will be written to the data file.
 - If you choose to use a test file, or if the window's data file does not yet exist, the window will write records to a temporary data file. When you exit Paint, the test file data is deleted.
5. When done, press [Esc] to return to the painting process for continued design and development.

If you create the window's data file at the end of the Paint session, you will have the opportunity to copy the test records into the data file.



Chapter F 5

Adding Advanced Features to Windows with Paint

This chapter describes how to use the advanced features of Paint to define more complex prompt and label characteristics. It includes additional information on the Prompt window, prompt and softkey options, and Paint menu options. For an introduction to painting windows in Advanced Revelation, see the “Creating Windows with Paint” chapter.

The Advanced Paint Options	F 5 . 1
Grouping and Reordering Prompts	F 5 . 2
Using Default and Duplicate Prompts	F 5 . 3
Defining Advanced Prompt Characteristics	F 5 . 4
Defining Smart Labels	F 5 . 11
Using Prompt Formulas	F 5 . 11
Displaying, Editing, and Updating Dictionaries	F 5 . 13
Data Validation	F 5 . 19
Data Formatting	F 5 . 27
Quick Painting Windows	F 5 . 32
Using the Paint Indexing Menu	F 5 . 33
Customizing Windows	F 5 . 33
Creating Custom Softkeys	F 5 . 39
Relating Windows	F 5 . 39
Setting Window Video Attributes	F 5 . 41
Establishing Window Security	F 5 . 41
Disabling Window Keys	F 5 . 43

Defining Collector Windows F 5 . 43

Adding Window Processing F 5 . 47

Developer Notes for Paint F 5 . 52



The Advanced Paint Options

The following table shows the way to access the Paint options covered in this chapter:

Option:	Location in Paint:
Prompt Window	Prompt option and softkey popups
Formula	Prompt option popup
Ditto	Prompt option popup
Smart label	Label option popup
Bindings	Softkey popup
Prompt Table	Softkey popup
Reorder	Softkey popup
Default Window	Softkey popup
List Dictionary	Softkey popup
Edit Dictionary	Softkey popup
Edit Help	Softkey popup
Groupings	Paint menu
Softkeys	Paint menu
Indexing	Paint menu
Quick paint	Paint menu
Customize	Paint menu
Set video	Paint menu
Processes	Paint menu
Security	Paint menu
Recalculate	Paint menu
Collector	Paint menu
Key Disable	Paint menu
Update	Paint menu

Each of these options is described in detail in the sections that follow.

Limits Imposed by Environmental Bonds

Advanced Revelation's application design tools will accommodate a variety of file structures. However, specific environmental bonds may impose restrictions that are not accommodated by a particular tool.

If you are developing an application that uses environmental bonding, please refer to the *Environmental Bonding* section. The documentation for each specific bond provides details about any limitations that should be taken into account when working with Advanced Revelation's Paint tools.

Grouping and Reordering Prompts

Prompts can have particular relationships with other prompts in a window. If you have several multivalued prompts, you can group two or more of those prompts together. You can also change the order in which the cursor moves from one prompt to another.

Associating Multivalued Prompts

When a prompt is multivalued, it can be grouped together with other multivalued prompts. That group of prompts is then permanently linked and an action taken on one prompt affects other prompts. For example, if you delete the first value at a grouped prompt, the first value at all the other associated multivalued (AMV) prompts will be deleted automatically.

Choose the **Groupings** option on the Paint Main menu to display the groupings popup. To group prompts together, type a group number and press [Enter]. Notice that the number you enter appears at the top of the popup, to the right of the AMV Group heading. Next, move the cursor to the first prompt in the group and press [Enter], to the second prompt in the group and press [Enter] again, and so on until you have identified all the prompts in the group. The group number appears next to each prompt name.

The popup acts as a toggle. A group number is assigned to or removed from a prompt by pressing [Enter] at the appropriate row in the popup.

To remove a group designation, type the assigned group number. Then press [Enter] at the appropriate rows in the popup.

- ☛ If you group a combination of vertical and horizontal multivalued prompts, all prompts in that group will default to the style of the controlling (first) prompt. For example, if you group two multivalued prompts – the first vertical, the second horizontal – both will be treated as vertical prompts when the window displays.

Changing Prompt Order

When you paint a window, the order in which you add prompts determines the order in which the cursor moves from one prompt to another when the window runs. If you are not in "Insert prompts" edit mode when you add a new prompt in the window, the new prompt is appended as the last prompt in the window. When the window runs, the cursor will move through all the prompts and then jump back up to the prompt you added.

You can change the prompt order with the prompt Reorder option. To display the Reorder popup, choose the **Reorder softkey** option, [Shift-F7]. The order in which prompts appear on this popup determines the order in which the cursor will move.

To move a prompt to a new location in the popup list, move the cursor to the entry for that prompt and press [Enter]. The prompt name appears in the Sort Column. Then move the cursor to the new location in the list with the [↑] and [↓] keys and press [Enter] again. The prompt name will move to the new location.

If a prompt is part of a group, its group number will appear in the AMV Group column on the Reorder popup. If you want to reorder associated multivalued prompts, use the Groupings option on the Paint Main Menu to remove the group number from those fields. Then use the Reorder popup to reorder the fields, and regroup the fields with the Groupings option.

Using Default and Duplicate Prompts

Paint lets you define a set of default prompt characteristics to use when creating prompts. It also allows you to duplicate characteristics you have defined for other prompts from the current window or from other windows.

Specifying Default Prompt Characteristics

To define a set of default prompt characteristics, choose the **Default Window** softkey option, [Shift-F8].

Paint has its own set of default characteristics for some of the Prompt window prompts. You can change these defaults and specify additional default values for the other prompts in the Prompt window. The characteristics you specify in this window become the new default setting. When you create a new prompt, Paint copies the default values specified in the Default Prompt window to the Prompt window for the new prompt and increments the *Position* number by one.

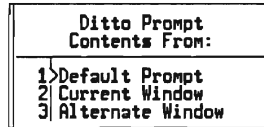
- ☛ Make your default specifications carefully. Each characteristic that you specify at the Default Prompt window will be used as the default for *all* prompts.

For more information on defining prompt characteristics, see “Defining Advanced Prompt and Label Characteristics” later in this chapter.

Duplicating the Contents of Another Prompt

Once you have defined a set of characteristics for a prompt in any window, you can copy the definition of that prompt and use it for another prompt. This can save a great deal of time when you have a number of prompts that are very similar.

To copy the definition of a prompt, position the cursor on a prompt and press [F4] to turn on edit mode. Then, press [F2] (Options) and select Ditto from the Prompt Options popup. The following popup will display:

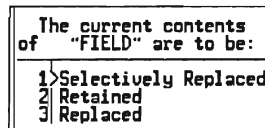


The first option copies the default definitions from the Default Prompt window, described in the previous section, to the current prompt.

The second option lets you choose a prompt from the current window to copy. It displays a list of all the prompts defined for the current window. Choose any prompt from the list.

The third option lets you choose a prompt from a different window to copy. You are asked for the name of the alternate window. After you specify a window name, you will see a list of all the prompts defined for that window. Choose a prompt to copy from the list.

Once you choose one of the three ditto options, the following popup displays:



If you choose the first option, **Selectively Replaced**, you are asked to confirm the replacement for each characteristic defined for the selected prompt using a popup. This allows you to replace some prompt characteristics but leave others unchanged.

The second option, **Retained**, lets you retain existing values for the current prompt and fill in only those characteristics that are currently blank with values from the specified prompt.

The third option, **Replaced**, replaces all the characteristics of the current prompt with the characteristics of the specified prompt.

Defining Advanced Prompt Characteristics

Advanced Revelation gives you many options for defining how a prompt behaves and how it displays information. This section describes how to use some of the more sophisticated prompt options. The basic options are described in the "Creating Windows with Paint" chapter.

Displaying the Prompt Window

The Prompt window lets you specify basic and advanced characteristics of the current prompt. You can define characteristics such as the name of the field associated with the prompt, the format for information displayed at the prompt entry line, special processing functions, and the relationship this prompt has with other prompts.

While you are editing a prompt, you can display the Prompt window. Position the cursor on the prompt or entry line that you want to define. If necessary, turn the editor on by pressing [F4]. Press [F6] (Softkeys) and choose the **Prompt Window** option from the Paint Softkeys popup, or use the Prompt Window softkey, [Shift-F6].

COMPANY Prompt		
Prompt Detail		
Field name	COMPANY	Entry type
Dict type	F	Key part
Data type	CHAR(20)	Multivalue
Position	1	Tab stop
Display length	20	Default
Justification	L	Row limit
Validation pattern		Edit mask
		Max length
Output format		Table mode length
Lower case		Table mode heading
Processing Code Command		
Options		
Detail help H	2	
Pre-prompt		
Post-prompt		
Invalid		

Defining Prompt Detail

Some prompts in the top part of the Prompt window let you define special prompt specifications.

Validation pattern Controls the kind of data that Advanced Revelation will accept at the prompt. For an explanation of how to make these specifications, see "Data Validation" later in this chapter.

Output format Controls the display format for unformatted information. For an explanation of how to make these specifications, see "Data Formatting" later in this chapter.

Key part Specifies which part of a multipart record key this prompt will accept and display. Each window has a key prompt, or prompts, which are used to enter or edit the key for each record. The key part specification is used to assign the part of the record key to which this prompt applies. For example, a multipart key might be used in a Parts file, where part one is the manufacturer's name and part two is the part number.

You can specify which part of the multipart record key this prompt will accept and display by typing a number. For example, "1" indicates the first part of the key field, "2" the second part, and so on. Do not make a key part specification for single-part keys and non-key fields.

- ☛ If you want to put symbolic fields between key parts, you must paint the window with the prompts in the proper order. You cannot use Prompt Insert mode or the Reorder option from the Paint softkeys to put a symbolic field between parts of the key.

Multivalue Indicates whether the field associated with this prompt is a single-lined, multivalued field. Normally you indicate that a field is multivalued by having two or more entry lines associated with a prompt. This option will usually be left blank, even for multivalued prompts. For single-lined, multivalued prompts, enter "1" for this specification.

Tab stop Indicates if a tab stop should be placed at this prompt. Tab stops allow users to jump to commonly used prompts by pressing the [Alt-T] key. For example, you might set a tab stop at the monthly payment amount field on a payroll window. That way the user can jump right to the payment amount field and skip the employee information fields which rarely need to be changed. Users can return to the previous tab by pressing [Alt-Y].

Default Specifies what action to take when a user presses [Enter] at a prompt without entering information. For example, you could fill in the system date automatically when the user presses [Enter] at a date prompt. Press [F2] (Options) at the *Default* prompt to display the Default Options popup.

Some of the most commonly used Default options are to duplicate data from a record entered previously or to fill in the current date and/or time.

There are two sequence counter options. A sequence counter increments a number by adding one to it. The **%S%** option is used to increment a number immediately every time you press [Enter] in a field that has this default specified. The **%SK%** option is used to increment sequential keys. It is commonly used for assigning record key numbers automatically. The record key number is incremented only when a new record is saved.

The **symbolic** option can be used when you want to display a value produced by a dictionary field at a prompt entry line. The product of a symbolic dictionary record is specified as a default by entering the name of the dictionary record in braces.

Subroutines are frequently used to produce a value to display in a prompt. You can use an R/BASIC external subroutine to calculate a value for the null default. In order to do this, create the program to calculate a value; when you've derived the appropriate value, load it into the system variable @ANS.

Row limit Allows you to specify the maximum number of rows of data that may be entered at a prompt when the prompt is part of an associated multivalued group. This specification is used only for associated multivalued fields. If no entry is made, there is no limit.

Edit mask Allows you to specify an edit mask character to appear at a prompt before the user enters information. Edit masks show how long the entry line is and what type of information to enter. For example, you might specify "###-##-####" for a social security number prompt. Edit masks are used only for display and perform no validation checks. Masks specified for individual prompts override those specified in the Customize window.

Max length Indicates the maximum number of characters that can be entered in a prompt. This is different from the display length, which specifies how many characters are displayed in the window. When the maximum number of characters is entered, an automatic carriage return is executed. The default is a maximum length of 65,534 characters.

Table mode length Indicates how much space is allowed for displaying information for this prompt in Table mode. Press [Ctrl-F5] (Table) when a window is running to display records in Table mode. If an entry is longer than the amount of space allocated for the prompt, only a portion of the entry will be visible in that cell in the table. While the cursor is in that cell, press [F3] (Zoom) to display the rest of the information.

Table mode heading Specifies the heading line to appear for this prompt when records are displayed in Table mode. If you do not specify a name, the name of the prompt is displayed.

Prompt Processing

You specify different types of prompt processing on the bottom half of the Prompt window's first page. Processing specifications let you define such characteristics as the options and help available for that prompt. The prompts used to specify processing are described below.

Each type of processing, if used, requires a code and a command entry. To see a list of available codes, press [F2] (Options) from any of the processing code prompts. For more information on codes and commands, see the "Using Codes and Commands" chapter.

- Options** Specifies what action will be taken when the user presses [F2] (Options) from this prompt. You specify a code and a corresponding command for each option. For example, a key field with a code of P and a command of @ID displays a list of record keys and returns a key from the list.
- Detail help** Allows you to display a help window automatically when the user presses [F1] (Context) from this prompt. Use H codes to define your help. There are a variety of H type codes available for supplying different kinds of help.
- Pre-prompt** Allows you to perform special processing before a prompt is displayed. For example, you might request a password before allowing data to be entered or changed. For more information, see "Adding Window Processing" at the end of this chapter.
- Post-prompt** Allows you to perform special processing after leaving a prompt. For example, in an invoice payment window you could compare the amount entered at the *PAYMENT* prompt to the *BALANCE DUE* amount and, if the payment amount is less than the balance due, add the difference to the *OVER 30* prompt. For more information, see "Adding Window Processing" at the end of this chapter.

Invalid Indicates a special action to take if the data typed at the prompt entry line does not match the defined validation pattern for the prompt. For example, you might display a message window. If no action is specified, the user will see a default message, and be repeatedly prompted until a valid entry is made. Press [F2] (Options) for a list of code and command options. See the “Using Codes and Commands” chapter for more information on codes and commands. For more information on validation patterns, see “Data Validation” later in this chapter.

Prompt Registers Stores information for use with different prompt code and command operations. These five buffers appear on the second page of the prompt window. For more information on using registers, see “Adding Window Processing” at the end of this chapter.

Joining Prompt Fields with Other Files

Windows can display and update information from more than one file at a time. In addition to the file assigned to a window, you can specify fields in different files that will be updated by entries made at the individual prompts in the window. A join field writes to a field in a record in another file.

By using joins, you can store information for a window in several different files. This permits you to create an efficient database structure.

For example, within a Customer window that updates the CUSTOMER file, you might include a *Notes* prompt that update a separate file used exclusively for storing customer notes. You might specify NOTES as the new file name, CUSTOMER_ID as the key, and “1” as the field’s position number at the Join prompts.

You establish a link between the field associated with a prompt and a field in another file using the Join prompts that appear on the second page of the Prompt window. While in the Prompt window with the Editor off, you can press [Pg Dn] to display the Join prompts.

File Specifies the name of the additional file to be joined to this window. This file will be updated with data entered at the prompt. Press [F2] (Options) to see a list of all attached files.

Key Specifies the name of the field in the current record that contains the key to the record in the additional file. This key is used to join the other file.

For example, the Customer window described above contains a *CUSTOMER ID* prompt. If you used a separate file for customer notes, you would specify CUSTOMER_ID as the key field so that every note you write would be associated with a particular customer.

Position Indicates the position number of the field in the other file that will store the information entered at this prompt. To determine the position, look the field up in the dictionary of the joined file. If the other file contains only one field, as a NOTES file might, you would specify “1” for the position number.

Delete Join Record Indicates if the joined record from the other file is to be deleted automatically when the associated record in this file is deleted. Again consider the NOTES file example. If you specified “yes” at the *Delete join record* prompt, whenever you deleted a customer record, the associated notes record would be deleted automatically.

Changed Key Delete Indicates whether to delete the associated record in a joined file if the value entered in the key field in the current file changes. You would enter “no” if you want to preserve the associated record in the joined file even though the value in the key field changes.

Displaying the Prompt Table

An alternative to using the Prompt window is the Prompt table. The prompt table displays the characteristics defined for all the prompts in the application window in a table format. Each cell in the table represents a prompt in the Prompt window. You can use the prompt table to edit the prompt characteristics of any existing prompt in the window. Table mode allows you to display popup options and get help just as you can in the prompt window.

To display the characteristics of all window prompts, choose the **Prompt Table** softkey option, [Shift-F5]. How the table appears depends on how many and what kind of prompts you have defined for your window. For example, here is the Prompt table for a Customer window:

Field name	Field	Data type	Field n	Prompt	F
CUST. NO	F	INTEGER	8	14	R
COMPANY	F	CHAR(20)	1	20	L
CONTACT	F	CHAR(20)	2	20	L
ADDRESS	F	CHAR(20)	3	20	L
CITY	F	CHAR(15)	4	15	L
ST	F	CHAR(2)	5	2	L
ZIP	F	CHAR(10)	6	10	L
PHONE	F	CHAR(15)	7	15	L

Defining Smart Labels

You can use a label to display such information as the system time and date automatically. Labels that display information are called Smart Labels. Choose the **Smart Label** option from the Label Options popup.

To define an action for the label, enter a code and command sequence. Press [F2] (Options) from the *Code* prompt for a list of available codes. Press [F9] (Save) when finished to save your entry. For more information on codes and commands, see the “Using Codes and Commands” chapter.

Using Prompt Formulas

A prompt can display the result of a calculation that uses values from other prompt fields. Formulas are useful for calculations such as finding the total for a number of items. A formula can be any R/BASIC expression.

R/BASIC expressions are commonly used for finding the total of values from several different prompts. For example, in an Invoice window, you might create an *Invoice Total* prompt that uses a formula to calculate the total of all the amounts entered at a multi-valued *Item Amount* prompt.

The result of the R/BASIC expression is loaded into a system variable called @ANS for “answer”. When you define a formula for a prompt, the value of @ANS is what is displayed at that prompt.

Using the Formula Builder

Advanced Revelation uses a feature called the Formula Builder to assist you in creating prompt formulas. When a formula is specified for a prompt, its associated field must be defined as a symbolic type in the data dictionary. The formula builder changes the field to a symbolic type (dictionary type S) automatically when you create your formula.

To start the formula builder, position the cursor on the desired prompt, press [F4] (Edit), and choose the **Formula** option from the Prompt popup. If the prompt is not defined as a symbolic type, you will be asked if you want to change types.

Once the formula builder window is displayed, press [F2] (Options) to display the first Formula popup.

Choose the formula type from this popup that matches the kind of formula you wish to build.

The formula builder walks you through the syntax for creating a single line R/BASIC program. Press [F2] (Options) from any lower case keyword in the syntax line to see a popup list of options for that key word. Depending on the options you choose at each point, you will see a different set of popups.

For more information on using the formula builder, see the “Building Formulas” chapter. For more information on R/BASIC expressions, see the *R/BASIC* section. For more information on defining data dictionaries, see “Displaying, Editing, and Updating Dictionaries” later in this chapter.

Specifying Window Recalculation

As an application window executes, it will normally recalculate all symbolic fields whenever the user enters data at a prompt. The window will do this for every symbolic field each time it executes, whether there were changes affecting a symbolic field or not.

In most cases, not every symbolic field in the window needs to be recalculated each time a prompt entry is made. For the sake of processing speed, it pays to avoid recalculating symbolics unnecessarily. So, in Paint you can specify when the recalculation of symbolic fields should take place. You can choose from several recalculating schemes, directing the window processor to use the method that best suits your window design.

The **Recalculate** option on the More Paint Functions menu calls the Recalculate popup. Each selection on the Recalculate popup recalculates the window's symbolic fields slightly differently.

- All** Immediately recalculates all symbolic prompts in the window each time values at other prompts change. This is the default setting for a window. For example, you might create an Invoice window that contains a multivalued *Item Amount* prompt used to enter individual line item amounts and a symbolic *Invoice Total* prompt that calculates the total. By selecting “All”, you direct the window processor to recalculate the invoice total after each item amount is entered.
- None** Recalculates the value of any symbolic prompt in the window when the cursor moves to that prompt. If you specified “None” for the example Invoice window described above, the invoice total would be recalculated only when the cursor moves to the *Invoice Total* prompt. Specifying “None” can save time in a window where each change might otherwise set off lengthy calculations.
- Custom** Lets you specify which symbolic fields to recalculate at what time. This option displays a list of all the prompts with symbolic fields for this window. Press [Enter] to toggle the message displayed beside each prompt from “only as used” (None) to “every change” (All) as desired.

Dependency Directs the window processor to calculate only the symbolic fields that are dependent upon the prompt entry that is currently being edited. This option provides top window processing speed while ensuring that the symbolic field data displayed in the window is accurate and up-to-date. Symbolic fields that are not dependent on data entered at other prompts in this window will be calculated and recalculated only when initially displayed or when they are accessed in the window's prompting order.

✎ Advanced developers can find additional information about the **Recalculate Dependency** option in the "Developer Notes for Paint" section at the end of this chapter.

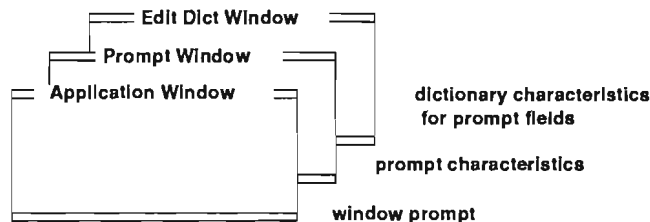
Displaying, Editing, and Updating Dictionaries

Each file associated with a window has a corresponding dictionary file that contains the file's field characteristics.

There is an intimate relationship between window prompts and dictionary field records. Both refer to the same data on file, although they may treat it differently.

Field and Prompt Characteristics

The field characteristics in the dictionary determine how information is stored in the file and how it is displayed in reports. The characteristics of the prompt determine how information is displayed in the window. The field characteristics specified in the dictionary definition may be used by the prompt. The specifications entered in the Prompt window are used only by that prompt in that application window.



You may define dictionary fields differently from prompts. You may also have special dictionary fields used for processing that do not appear in a window. For more detailed information on dictionaries, see the "Advanced Revelation Dictionaries" chapter in *Filing Systems*.

Using Different Binding Methods

The relationship between creating the prompts in a window and creating and updating field records in the dictionary is determined by the binding mode you select for the window.

When the prompt definition is written to the dictionary, it updates the dictionary record for that prompt.

The top portion of the Prompt window displays the definition for the specified prompt. The first eight prompts in the Prompt window are directly tied to the data dictionary through the binding process. If the dictionary record exists, then the specifications are copied to the Prompt window. If you are entering a new prompt, and the dictionary record does not exist, the prompt definition is used to create a dictionary record unless binding is off.

Specifically, the prompt entries used to create a dictionary record are:

- Field name
- Dict type
- Data type
- Position
- Display length
- Justification
- Validation pattern
- Output format

Changing Your Binding Method

The default binding method in Paint is Autobind. To change your binding method, use softkey [Shift-F2]. This displays a popup of available binding methods.

Auto This method is commonly used when you are defining a new file while you design your window. Use this binding method to automatically build dictionary records as you enter window prompts.

When you enter a new prompt, Paint uses a set of dictionary defaults to create a new record in the data dictionary for the field associated with that prompt. The field name is the text of the prompt you entered. The fields are numbered consecutively in the order that you enter the prompts.

If the text of the prompt label matches an existing field name in the dictionary, Paint will use the existing dictionary information for the prompt definitions.

Verify Use this binding method to verify your prompt entry against existing records in the dictionary before creating a new dictionary record. This method is commonly used when some dictionary items exist and you want to associate some of your prompts with existing dictionary records.

When you enter a prompt, Paint checks to see if the text of the prompt label matches an existing field name in the dictionary. If it does, the Verified popup displays. You have two options. You can bind the prompt to the field of the same name and use the information in that dictionary record as the prompt definition. Alternately, you can choose to associate the prompt with a different field.

If the text of the prompt label does not match an existing field name in the dictionary, the Not Verified popup displays. You can create a new field just as you would in Autobind or choose to associate the prompt with a field of a different name. When you choose Option 2 to bind the prompt to a different field, a popup list of all the current fields in the data dictionary file displays. Select the field you want to use.

Off Use this binding method when your entire dictionary has already been defined, or when you want to create each dictionary entry yourself. The relationship between the prompts and the dictionary records will be determined completely by the designer. This mode is typically used when the dictionary has been defined for the application window's associated file.

In this mode, when you enter a new prompt Paint does not create a new field record in the dictionary. The dictionary is not updated.

If the text of the prompt label matches an existing field name in the dictionary, Paint will read the existing dictionary record and use the dictionary information for the prompt definitions.

You can use the Quick Paint option to paint prompts if your dictionary exists already. The binding mode in Quick Paint defaults to Off. For more information, see "Quick Painting Windows" later in this chapter.

Number This binding method is automatically used by Paint when you create a Collector window. Paint numbers your prompts but makes no dictionary entry, since Collector windows are not associated with dictionaries. Number binding is used to assign the system variable @RECORD a field position for each prompt. For information on collector windows, see "Defining Collector Windows" later in this chapter.

Listing Dictionary Fields

To see a list of all the dictionary fields for the file associated with your window, choose the **List Dictionary** softkey option, [Shift-F9].

The List Dictionary window shows the name, position, and type for each associated dictionary field. You can change these characteristics with the **Edit Dictionary** option described below.

Editing Dictionary Fields

To edit any of the dictionary characteristics of a field, choose the **Edit Dictionary** softkey option, [Shift-F10]. It displays the Dictionary window. Fill in the name of the field to edit. If the Editor is on, Paint will fill in the current field name automatically.

Changing field characteristics in the dictionary (for example, display length) will not automatically update the same characteristics for a prompt in a window. If you want to synchronize prompt characteristics with field characteristics, you will need to make separate changes to the prompt characteristics for the window using Paint. See “Synchronizing Field and Prompt Characteristics” later in this chapter.

- ☛ If you change a data type specification in the dictionary, be sure to use Paint to make the same change to any window prompts that reference that field.

Deleting Dictionary Fields

Use the Dictionary window to delete fields from a file's dictionary:

- At the *Field name* prompt, enter the name of the field to delete.
- When the field definition displays, press [Alt-D] two times to delete the field.

During a Paint prototyping session, you may add and remove prompts from a window. However, the field definition associated with a window prompt is always preserved in the dictionary, even if you delete the prompt from the window. Unless you delete the field definition, it becomes part of the record structure. When you make the data file, *all* of the fields currently defined in the file's dictionary become part of the record structure.

It is advisable to remove all unwanted field definitions from the dictionary before using Paint to create the data file. If you don't, you will create fields in the data file that you do not intend to reference. When working with bonded file types, such as dBASE and ASCII, these fields may take up extra room in your data file, inflating the size of the file unnecessarily.

Environmental Bond Limitation

Each field in the dictionary references a position in the record structure. The field's record position is identified by number.

Environmental Bonds require that all of the field positions in the record structure *must be sequential*. If you create a gap in the sequential field positions by deleting a field, you must decrement by one the position numbers of the fields that follow the deleted field. When you attempt to create a data file, using a dictionary in which the field positions are not sequential, the bond will prevent the file creation and display an error message.

For example, a CUSTOMER file might have the following fields in positions 0 through 3:

Field Name	Position
COMPANY_NAME	0 (key)
CONTACT	1
ADDRESS	2
PHONE	3

If you delete the CONTACT field, there is no longer a field assigned to position 1. This creates a gap in the sequential position numbers assigned to each field.

Field Name	Position
COMPANY_NAME	0 (key)
ADDRESS	2
PHONE	3

To eliminate the gap, you must change the positions assigned to the fields that follow position 1 in the record structure — in this case, you would change ADDRESS to position 1 and PHONE to position 2.

To determine the current positions of the fields in the dictionary, use the List Dictionary softkey [Shift-F9].

If necessary, use the Dictionary window to change a field's position number:

- At the *Field name* prompt, enter the name of the field to delete.
- When the field definition displays, move the cursor to the Position prompt and assign a new position number to the field.

For more information about field positions see the "Introduction to Bonding" chapter in *Environmental Bonding*.

Synchronizing Field and Prompt Characteristics

During a painting session, your edits to prompt characteristics may cause the current prompt characteristics and the associated field dictionary record to disagree. You can use Update to synchronize these elements.

The Update feature allows you to update certain prompt characteristics with dictionary field definitions and vice versa. The update occurs when the prompts are edited and when you press [F9] (Save) to save the window template. These are the characteristics that can be updated:

- Dict Type
- Data Type
- Position
- Key Part
- Display Name
- S/M (Single or Multivalue)
- Output Format
- Justification
- Length
- Validation Pattern

To make use of the update feature, choose the Update option on the More Paint Functions menu to display the following popup:

Select Update Method
1>Update all prompts from dictionary
2>Update dictionary from all prompts
3>Select prompts to update from dictionary
4>Select prompts to update dictionary

Depending upon the update method selected from the popup, either some or all of the current characteristics for the prompt will be written to the field definitions or some or all of the current definitions for the field will be written to the prompt characteristics.

Update all prompts from dictionary

For all prompts in the window, updates some or all of the prompt characteristics noted in the list above with the corresponding dictionary definitions.

Update dictionary from all prompts

For all prompts in the window, updates the associated dictionary field characteristics with some or all of the prompt characteristics noted in the list above.

Select prompts to update from dictionary

Lets you choose prompts in the window to update from the dictionary. For the prompts selected, updates some or all of the prompt characteristics noted in the list above with the corresponding dictionary definitions.

Select prompts to update dictionary

Lets you choose prompts in the window to use to update the corresponding field definitions in the dictionary. For prompts selected, updates the associated field characteristics with some or all of the prompt characteristics noted in the list above.

Data Validation

Advanced Revelation gives you great flexibility in being able to check the validity of data input by a user. Data validation can be defined in the Prompt window in Paint and in the Dictionary window. This will help you to maintain the integrity of your database by preventing invalid data from being entered into the system.

Validation Patterns

A validation pattern is a format against which input data is checked. Validation patterns are specified at the *Validation pattern* prompt in the Prompt window and in the Dictionary. If the data matches the prescribed format, the data is considered good, and is allowed. Data not matching the format is rejected, and the user sees a message that the data did not meet editing criteria. A list of accepted formats is also displayed. The user is then given another opportunity to enter the data.

For example, you might use a validation pattern at a field for ZIP codes. A validation pattern would guarantee that only 5-digit (for old U.S. code) ZIP codes could be entered into the system.

- ☛ Each time you make a data type specification for a new prompt, a validation pattern corresponding to your data type specification is created automatically. The validation pattern displays at the *Validation pattern* prompt in the Prompt window and in the Dictionary window. When the data entry window updates a foreign file type (like dBASE or ASCII), the validation pattern ensures that the data entered at the prompt is of the right data type.

Types of Validation Patterns

There are several different types of validation patterns available in Advanced Revelation. These types are described below.

Literal Matches

Literal matches allow you to check if the data input by a user matches certain literal text. This is useful if you have a limited number of words or other strings that will be acceptable in the field.

For example, if you were establishing a pattern match for a field into which school grades were being entered, it might be appropriate to limit the entry to only the letters A through F (less E).

To establish a pattern for a literal, enclose the literal text in single or double quotes. In the above example, there would be five entries: "A", "B", "C", "D", and "F". As long as the input data met any one of these, it would be accepted into the field.

Pattern Matches

If the data input by the user is to meet a generic format such as "all numeric" or "2 alphabetic characters," you can use a pattern match. Advanced Revelation provides these generic patterns:

Pattern	Meaning
<i>n</i> A	<i>n</i> number of alphabetic characters (A-Z, a-z)
<i>n</i> N	<i>n</i> number of numeric characters (0-9)
<i>n</i> X	<i>n</i> number of any character
<i>n</i> Z	up to <i>n</i> characters of any type

The *n* in these patterns is filled in with the proper number of acceptable characters. The number zero is used to mean "any number of these."

For example, the pattern match "3A" means "3 alphabetic characters," the pattern "4X" means "4 of any kind of character," and the pattern "0N" means "any number of numeric characters."

Ranges

When dealing with numeric data, you may wish to limit input data to a certain range. You can do this by specifying a starting and ending value for the data.

The format for a range check is:

Format	Meaning
(<i>n</i>)	All values less than or equal to the number <i>n</i> are valid
(<i>n,m</i>)	All values between <i>n</i> and (<i>m</i>) are valid.

For example, the range check "(500,9999)" would ensure that the value entered in the field would be between 500 and 9,999 (inclusive).

Conversions Certain types of data in Advanced Revelation are converted into an internal format for storage efficiency and ease of manipulation. These types are dates, time (e.g. 04:00 AM), boolean strings, and decimal numbers, including monetary amounts.

For more information about input conversion patterns see ICONV in the chapter called “R/BASIC Command Reference” in the *R/BASIC* section.

The conversion validation pattern allows you to check that data input by a user meets one of these types. The formats of these validation patterns are:

Format	Meaning
(B)	The data must match the specified boolean string format (default is Y or N)
(D)	The data must be a valid date format
(MT)	The data must be a valid time designation
(DT)	The data must be a valid date and time designation. Either the date or time may be delimited with spaces, but not both.
(MD n)	The data must be a decimal number with n decimal places

Advanced Revelation is quite liberal in its interpretation of data to be converted. For example, these are just some of the valid formats in which a date can be entered:

```
01/01/89
JAN 01 1989
01 JAN 89
010189
01-01-89
```

If you specify a conversion validation pattern, Advanced Revelation will strip out extraneous information and normalize the data. For example, even though the user may enter the date in the form “Jan 01, 1989”, Advanced Revelation will convert it to a standard format such as “01-01-89” (the exact format is defined at the *Output format* prompt in the Prompt window). If you specify the pattern “MD2” to mean that the data must have two decimal places, Advanced Revelation will insure that there are two even if the user only enters one.

There are other conversion patterns available, but these are used infrequently. Additional conversions enable you to check for the valid input of hex, octal, or binary data.

- ☞ Conversions will both check for valid data as well as actually change the data into internal format (as long as the input data is valid). Because the data will be converted immediately, conversion validation patterns are almost always used in combination with formatting specifications to display the data properly. For more information about internal formats and formatting data for output, see “Data Formatting” elsewhere in this chapter.

Uniqueness Test for Multivalues

The uniqueness validation pattern permits you to specify that entries in a multivalued prompt must be unique. This functionality will help you when you must guarantee that no duplicate values be entered at a multivalued prompt.

To establish a uniqueness pattern, use the code %U% (percent signs surrounding a capital U).

When a user enters data at this multivalued prompt, the window will check to see if the same data has already been entered at that prompt for that record. If not, the data is accepted. If the data is not unique, a message will display which informs the user that entries at that prompt must be unique.

Verifile

The Verifile validation pattern checks the validity of input data by looking it up in another file. The input data must be (or must not be) the key to a record in another file. A specified file is checked in order to determine if the data matches an existing record key.

Verifile validation patterns are useful when the data that is valid might change over the course of time, or when you are entering data that relates to another file. An example might be product codes used in a sales order application. When you enter a product code at the *Product Code* prompt in an Invoice Entry window, you can use a Verifile validation pattern to confirm that the product code is valid by looking it up in a PRODUCT file.

The format of a Verifile is:

```
<FILENAME,{type},{concat.literal}, ⇨  
{code,command}>
```


The argument in the Verifile command must follow the prescribed order, using commas as delimiters. If an argument will not be used, a comma is still entered in the Verifile command line to maintain its position.

The data entered at the prompt is used as a record key to read a record in the file specified by filename.

The optional *type* N (not) indicates that the typed information will pass the test if it is *not* found in the file. If *type* is null or another character, then the typed information will pass the test if it is found in the file. For example, the verifile **<PRODUCTS>** means that any data entered into that field must be the key to a record in the PRODUCTS file. On the other hand, the verifile syntax **<PRODUCTS,N>** means that any data entered into that field must *not* be the key to a record in the PRODUCTS file.

The optional *concat.literal* is concatenated to the front or back of the value typed in the entry line. If the : (colon) is on the front of the literal then the literal is placed at the end of the value. If the : (colon) is in the back of the literal then the literal is placed at the front of the value.

The optional *code* and *command* allows you to specify the action to take following a Verifile check when the data fails the test. This *code* and *command* must set the system function STATUS (). Therefore, your code must be S to call a subroutine. For more information, see "Using an R/BASIC Program to Validate Data" below.

User-Specified Matches

Occasionally, you cannot check the validity of input data by one of the above methods (or by combining them — see below). In that case, you can define your own validation pattern routine using a symbolic dictionary item, an R/BASIC subroutine, or a code and command.

For example, suppose you are painting a window for an invoice. You wish to verify that the date for the field DUE_DATE does not precede the date for the field INVOICE_DATE. Since none of the other validation methods work, you will need to define your own logic to do this.

You can call your own process using one of the following methods:

Process	Description
@code command@	Specifies a code and command process used to verify the data.
[subroutine]	Specifies a subroutine that performs a user defined conversion. For information on using this special type of subroutine, see the "R/BASIC System Subroutines" chapter in the <i>R / BASIC</i> section.
{dictionary.name}	Specifies a symbolic dictionary item that verifies the data.

The data must meet criteria specified by one of these operations.

Using an R/BASIC Program to Validate Data

When you run a subroutine or make a dictionary call to validate data, your routine should read the data from @ANS, perform the validation check, and then set the STATUS() function based on the contents of @ANS. The STATUS() function is read by the Window interpreter to determine what action to take next:

Status	Action
0	Good pattern (this is the default)
1	Bad data. Print system error message and reprompt.
2	Bad conversion. Print system error message and reprompt.
3	Reprompt (no error message)

Linking Multiple Patterns with an OR

A feature of Advanced Revelation validation patterns is that you can specify several different formats for a single field. So long as the data meets any one of the patterns, it is considered valid. To enter multiple patterns with an implied OR, enter each pattern on a new entry line at the prompt. For example, a ZIP code field might allow either a 5-digit code or a 9-digit code, since both of these are valid.

Linking Multiple Patterns with an AND

Advanced Revelation also permits you to combine more than one validation pattern into a single check. This enables you to specify that the data must meet several criteria to be valid. To enter multiple patterns with an implied AND, enter each pattern on the same entry line, separating each pattern with a + (plus), subvalue mark, or _ (underscore). For example, you might specify that data is both to be a valid date and to fall within a specified range.

Different types of validation patterns can also be strung together ("concatenated"). For example, you can check that an invoice number always begin with the literal "AA" followed by up to 5 numeric characters.

Concatenating Validation Patterns

Advanced Revelation enables you to string together different validation patterns as required. For example, you can concatenate literals and pattern matches within a single validation pattern.

These validation patterns are typical for phone number fields:

3N'-3N'-4N

Accepts three numerics followed by a dash, followed by three more numerics, followed by another dash, followed by four numerics

'(3N)' 3N'-4N

Accepts the same as above, but requires parentheses around the first three numerics, and a space separating the first three from the second three numerics

The validation patterns use both literal text (enclosed in quotes) and pattern matches that specify the number and type of the acceptable characters.

Combining Validation Patterns

You can also combine different validation patterns when you want the combination of the two to function as a single validation pattern. If you do this, the input data must meet both validation patterns before it is considered valid.

For example, you might wish to create a validation pattern that checks for up to six numeric characters. The pattern match "0N" (any number of numerics) is not adequate, nor is the pattern match "6Z" (up to six of any character). The combination of the two, however, creates the proper validation pattern.

To combine validation patterns into one match, use the underline () or plus (+) symbol. For the example above, this would be the format for the validation pattern:

```
0N+6Z
```

```
0N_6Z
```

Both mean “any number of numeric characters up to 6.”

Range checks can be combined with conversions to put a limit around the values that the conversion will accept. It’s important that the conversion be specified as the first part of the pattern match expression. For example, the following pattern combines a conversion for a two-place decimal number with a range check:

```
(MD2)_ (500,1000)
```

This validation pattern accepts any number between 5.00 and 10.00, inclusive. In this example, the number 10.001 would pass, since the conversion will truncate any decimal places past the second one.

- ☛ The conversion of input data is done *before* the range is checked. To check for a valid range of dates, use the internal (integer) format of the date in the range check. There is more information about internal formats in the “Data Formatting” section later in this chapter.

Creating a Popup of Validation Patterns

You can create a popup of the validation patterns you have defined. This display-type popup is called with the [F2] key if no other *Options* have been defined for the prompt. The validation patterns and the popup that displays them can be defined at the same time. The popup will return a single selection at the prompt. If desired, you can create a display only type popup that does not return a value.

- ☛ Single-selection popups will be most useful at prompts where the validation pattern specification is for literal values. When a single-selection popup is used, the user can select the appropriate value, which is returned at the prompt. Display-only type popups are more appropriate at prompts where the validation pattern specifications are for conversions or pattern matches.

Defining the Validation Pattern Popup

To create a single-selection popup, define the validation patterns normally. Then add a | (vertical bar) to the end of each individual pattern, and follow the bar with a description that is to appear in the popup. The validation pattern that is specified before the | is returned at the prompt.

For example, if your validation pattern dictates that the prompt will accept valid letter grades (A, B, etc.), you can enter the *Validation pattern* values like this:

```
"A" | Excellent
"B" | Above Average
"C" | Average
"D" | Below Average
"F" | Failing
```

If a user presses the [F2] key at this prompt, a popup appears. The popup displays the letter values in one column and the descriptions in another. The user can choose the appropriate pattern, which is returned at the prompt.

To make the Validation Pattern popup a display only type, add a | (vertical bar) at the end of the last validation pattern description. For example:

```
"A" | Excellent
"B" | Above Average
"C" | Average
"D" | Below Average
"F" | Failing|
```

If a user presses the [F2] key at this prompt, and if no other *Options* have been defined for that prompt, a popup appears. This is a display-type popup, which will not return a value to the prompt.

If a code and command has been specified at *Options* for this prompt, the Validation Pattern popup will not display.

Data Formatting

Advanced Revelation permits you to specify formatting characteristics for data on file. This capability is used primarily for data that is stored differently from the way you wish to display it.

Although the storage of Advanced Revelation data is inherently efficient, you may wish to store certain types of data in a compact or converted format. This is typical with data concerning dates, time, or decimal numbers and money, or with data such as phone numbers or social security numbers that are displayed in a fixed format.

This section discusses the means by which you can define the format of data for display. For details about the conversion of data into internal formats, see the "Data Validation" section in this chapter.

Types of Formatting

There are two basic types of output formatting available for data: conversions and masking.

Conversions format data according to one of a number of predefined, generic conversion names. For example, dates are formatted by specifying one of a set of special date conversion names such as "D" or "D2-".

Masks are strings of special characters that represent the actual layout of the display field. For example, a display mask for social security numbers might look like this:

L###-##-####.

Output formats are specified at the *Output format* prompt in the Prompt window and in the Dictionary.

Formatting with Conversions

To cause Advanced Revelation to format data using a conversion, you simply need to specify the proper conversion name. Conversions are available to format date information, time information, boolean data, and decimal number or monetary data.

- For more information about the use of conversions, see the OCONV entry in the "R/BASIC Command Reference" chapter in the *R/BASIC* section.

The format for the name of the date conversion is:

Format	Description
Dnx	D specifies date conversion, <i>n</i> specifies the number of digits to display for the year, and <i>x</i> specifies the character to use as the m/d/y delimiter
Da	D specifies date conversion, <i>a</i> specifies the code indicating the arrangement of m/d/y (see chart below)

Dates The following table illustrates the name of each date conversion, the format it represents, and a sample of data displayed for each. Each conversion is based on an input date of April 1, 1989.

Table F5-1
Date Conversions

Name	Format	Output Example
D2-	mm-dd-yy	04-01-89
D4-	mm-dd-yyyy	04-01-1989
D2/	mm/dd/yy	04/01/89
D2/E	dd/mm/yy	01/04/89
D2/J	yy/mm/dd	89/04/01
DE	dd MMM yyyy	01 APR 1989
DF	MMM yyyy dd	APR 1989 01
DG	yyyy dd MMM	1989 01 APR
DH	MMM dd yyyy	APR 01 1989
DI	dd yyyy MMM	01 1989 APR
D2J	yy MMM dd	89 APR 01

Times Conversions for time are flagged with the code “MT”. The following table illustrates variations on the display of 2:00PM:

Table F5-2
Time Conversions

Name	Format	Output Example
MT	HH:MM	14:00
MTH	HH:MMmm	02:00PM
MTS	HH:MM:SS	14:00:00
MTHS	HH:MM:SSmm	02:00:00PM

Date/Times Conversions for a combined date and time are flagged with the code “DT”. The syntax for a date/time conversion is:

DT{<date_conversion>}{^{nc}⇨
{<time_conversion>}

- where:
- <date_conversion> is any valid date conversion
 - ^ delimits the date conversion from the time conversion

n is the number of characters between the date and time strings. Default is one.

c is the character to be used to separate the date and time strings. Default is space. To avoid ambiguity, if *n* is specified, *c* is required unless the time specifier starts with H or S.

<*time_conversion*> is any valid time conversion

For example, to obtain an output of:

APR 01 1989***12:00:00PM

use the output pattern:

DTH^3*HS

Decimal Numbers and Money Decimal number and monetary conversions all begin with the letters “MD”. The general format is:

MD*n**x*

MD specifies decimal conversion, *n* specifies the number of decimal places, and *x* specifies any character(s) that are to be included in the conversion ('\$', commas, etc.)

The following table illustrates variations on the decimal conversion.

Table F5-3
Decimal Numbers and Money Conversions

Name	Format	Output Example
MD2	nnnn.nn	1234.56
MD4	nnnn.nnnn	1234.5678
MD2,	n,nnn.nn	1,234.56
MD2,\$	\$n,nnn.nn	\$1,234.56
MD2,-	\$n,nnn.nn-	\$1,234.56-

Formatting with Masks

Masks use pound signs (#) to mark a place for each character of the displayed data. These pound signs are combined with other characters (for example, punctuation) and justification information to create a full mask.

A mask always begins with a letter designating how the data is to be justified within the display area created by the mask. Justification codes are:

Justification Code	Meaning
L	Left-justify within the display field
R	Right-justify within the display field
C	Center the data within the display field

Any characters can follow the justification code. If they are pound signs, they will be replaced with data (or spaces). Any other non-numeric characters will be treated as literals and embedded into the display field.

When the data from the field is displayed, each pound sign in the mask is replaced with a character from the field. If the field is too long for the mask, the display of data is truncated. If the mask is longer than the data in the field, the mask will be padded with spaces.

The following table illustrates some sample masks and the output they produce:

Table F5-4
Conversion Mask Samples

Data	Mask	Output
AA345	L##-###	AA-345
23	R####	23
345	C#####	345
2066439898	L(###)###-####	(206) 643-9898
ABCDEFGHI	R#-###-###	C-DEF-GHI

User-Specified Formats

Occasionally, you cannot achieve the correct data format by one of the above methods. In that case, you can define your own format routine using a code and command.

For example, suppose you are painting a window for an invoice. You wish to format the date using the entire word for the month. Since none of the format methods work, you will need to define your own logic to do this.

You can call your own process using the following method:

```
@code command{,argument}@
```

This format defines a code and command to call your process. For instance, the code S and the command `FORMAT_DT` calls the subroutine `FORMAT_DT`. This method is intended for the advanced user and requires the use of the system variable `@ANS`.

Internal Formats

Advanced Revelation stores the contents of certain fields in an internal format. These fields are those that contain data pertaining to dates, time, boolean strings, and decimal numbers, including monetary amounts.

The internal format for each of these data types is actually an integer value. The internal format for a date is the number of days before or after December 31, 1967. The internal format for a time is the number of seconds after midnight. The internal format for a boolean string is 1 (true) or 0 (false). The internal format for a number is the number without the decimal place or other punctuation.

Advanced Revelation uses internal formats for several reasons. The internal format by which Advanced Revelation stores this data takes up less storage space, since it excludes punctuation and other extraneous characters. This makes the file more efficient.

Internal formats also make it easy to manipulate the data in these fields. For example, the internal format for dates makes it very simple to calculate elapsed time (as required for aged invoice reports, for example) without ever needing to account for unequal numbers of days in a month, leap years, etc.

In addition, the use of internal formats helps to normalize the data. Rather than depending on data entry personnel to enter dates or times in a consistent format, Advanced Revelation can create a standard output format based on the internal format of the data.

Internal formats need to be converted for display, however. The various conversions discussed under "Data Formatting" in the preceding section provide you with the means to convert from internal to external formats automatically.

Quick Painting Windows

If you have already defined a dictionary for the file associated with a window, you can use Quick Paint to create window prompts. Quick Paint is available as an option on the Paint Main menu.

Quick Paint allows you to choose one or more fields from an existing dictionary, and then have Paint automatically create prompts in the window. This saves you the trouble of having to create individual prompts and bind them to the dictionary. You normally Quick Paint a window when you first create it. However, you can also use Quick Paint to add prompts later.

Displaying a List of Fields

To display a list of fields for the file associated with the current window, choose the **Quick Paint** option from the Paint Main menu.

To choose fields to display in your window, move the cursor to the desired field name, or type in the number that appears to the left of the field name, and press [Enter]. Select the fields in the order you want them to appear in your window. As you make your choices, notice that numbers indicating the order of the selected fields appears in the far right column on the popup. To cancel a field selection, again move to the field name or type in the number to the left of the field name and press [Enter].

- When quick painting, always be sure you choose the key field first. This field is identified as 0 (zero) in the Number column on the popup. The key field must be the first prompt in a window.

Once you have selected all the fields you want, press [F9] to save your choices.

When you return to the window, you will see that Paint has automatically added a prompt for each field in the specified order and labeled it with the field name. To change the prompt name that appears on the window, move the cursor to that name and press [F4] to start the Editor. Then enter the new name.

Once a prompt has been painted in the window, you can use any of the prompt options available in Paint to change the way it looks and behaves. You can also use options on the Paint menus to customize the window.

Using the Paint Indexing Menu

You can index fields in a file using the Indexing option on the More Paint Functions menu or the File Tools Menu.

To display the Indexing menu, choose the **Indexing** option on the More Paint Functions menu.

This menu has the same options as the Indexing menu available from the File Tools Menu. For more information on using these menu options, see the “Creating and Maintaining Indexes” chapter.

Customizing Windows

Prompts on the Customize window allow you to change how your window looks and behaves as a whole and to define a general window and prompt help structure.

Displaying the Customize Window

To display the Customize window, choose the Customize option from the Paint Main Menu.

The first page of the window contains prompts that let you define general characteristics of the window, protect and preserve window information, and add two kinds of window help. The second page contains special registers used for window processing.

Specifying Window Characteristics

Descriptions of the prompts used to define general window characteristics follow.

Border type Allows you to specify five different styles of window borders. Press [F2] (Options) to display a popup of available border types. The default entry is border type three.

Locking Indicates the type of locking you want for records displayed in the window. Locking is a way of protecting information when the database is accessed by more than one user at a time. If your application will run on a network, record locking should be used to protect your data.

Five window locking options are available in Advanced Revelation: no locking, shared lock, exclusive lock, exclusive coordinated lock, and shared coordinated lock.

☛ These locking options are only functional on networks that support these kinds of locks. See your network manual to determine which types of locking are available to you.

Press [F2] (Options) to display a popup list of locking options. To choose an option, position the cursor and press [Enter].

Exclusive Lock (the default locking method)

Exclusive locking means that the first user who calls a record obtains exclusive write privileges for that record. Under an exclusive lock, other users can display a record while it is locked, but they cannot update it. A locked out user will see messages indicating that the record is in use elsewhere, that the displayed data is protected, and that any changes made cannot be saved. The locked out user will have to reaccess the record when it is no longer being used by the user who has set the lock.

☛ When a window is painted, if no other type of locking is specified, exclusive record locking is implemented.

No Locking

If no locking is specified for the window, the window can be used to access and update any record in the associated data file. No locking means that when a read or write occurs, no check will be made to see if any other process is using the record or has attempted to lock the record. It allows access to a locked record, and allows that record to be overwritten in spite of the lock. This can have a serious impact on data integrity in a network environment. No lock is rarely specified, unless you are certain that you will never run the application on a network.

Shared Lock

Shared locking is intended solely for use in “view only” windows where the window itself cannot update records. When used in such a window, shared locking prevents anyone from putting an exclusive lock on the current record while it is displayed. Effectively, this prevents any user with file update capabilities from updating the record while it is displayed in a “view only” window.

For example, an accounting system could be set up with both exclusive and shared locking. The accounts receivable clerk uses a data entry Customer Account window to enter and update the accounting records in the CUSTOMER file. Updates to records in the file require an exclusive lock. A group of customer service representatives use a “view only” Customer Information window to display accounting records from the CUSTOMER file. Specifying shared locking in the Customer Information window prevents the accounts receivable clerk from updating a record in the CUSTOMER file while that record is displayed in the Customer Information window. The shared lock insures that the data viewed by the customer service representative remains unchanged during the time that the record is displayed.

☞ When shared locking is specified for a window, the window should be used only to display data. Make it a “view only” window by either disabling the [F9] (Save) key or substituting some other process for the write process.

**Exclusive
Coordinated Lock**

An exclusive coordinated lock operates like an exclusive lock with one added function: before locking the record a check is made to see if the data file itself is locked. Exclusive coordinated locking is used in a window when some process in the application may lock all records in the associated data file. With this type of locking, if the file is locked, the record cannot be locked. Conversely, if the record is locked, the file cannot be locked.

- ☞ A process that places a lock on all records in a data file only recognizes *coordinated* locks. You must use an exclusive coordinated lock whenever the window updates a data file that is sometimes used in a process that employs file locking. This is the only way to ensure that you can obtain an exclusive lock on the record when using the application window. Under an exclusive coordinated lock, while the record is locked, other users can only display the record and processes that lock the entire file cannot be run until the lock is released.

For example, the Batch Update process uses an exclusive file lock. Therefore, if your application window updates a file that is accessed by the Batch Update process, the window should employ exclusive coordinated locks. A window with an exclusive coordinated lock will recognize when a file lock has been set by another process. This prevents record modification by the window during a batch process.

Shared Coordinated Lock

Shared coordinated locking should be used in “view only” windows to prevent a process from locking the data file while the shared lock on the current record is in operation. Shared locking will prevent an exclusive coordinated lock. No process that uses exclusive coordinated locks can update the file while a shared coordinated lock is in effect on the file.

- ☞ A process that places a lock on all records in a data file only recognizes *coordinated* locks. You must use shared coordinated locking whenever the “view only” window accesses a data file that is sometimes used in a process that employs file locking (for example, a Batch Update process). This is the only way to ensure that the current record remains unchanged during the time that it is on display in the application window. Under a shared coordinated lock, while the record is locked, other users can only display the record. Processes that lock the entire file or require an exclusive lock cannot be run until the shared coordinated lock is released.

Menu

Names a menu to display instead of the Main Menu when the user presses [F10] while running the window. Enter the name of the appropriate menu template stored in the MENUS file.

Application

Names the application this window is associated with. This entry is a comment and does not affect the way the window behaves.

Exploding window

Indicates a special method of displaying a window. The window builds itself from the inside out when displayed on the screen, so it appears to explode.

- Start prompt** Indicates the number of the prompt the cursor will move to after the record is displayed. The user must first enter a value in the key prompt. Then the cursor will jump to the *Start* prompt. This entry line is normally left blank.
- Data/Dict** Indicates whether the Data or Dictionary portion of a file will be updated by the window. This is for your information only. You cannot make a specification at this prompt. Normally the prompt remains blank, indicating that the window updates a data file. A "D" at the prompt indicates that the window updates a dictionary file.
- Default mask** Specifies a character or set of characters to fill in on the entry line for every prompt. Mask characters help the user determine the length of each entry line and what kind of information to enter.
- If you enter a mask, the specified characters are filled in each time the cursor moves to a new entry line. Make your mask as long as your longest prompt.
- You can also specify a mask for an individual field with the prompt window. Individual prompt masks override default window masks.
- Registers 1-10** Store information associated with a window while running a window process. The window registers are located on the second page of the Customize window. For more information on window processing, see "Adding Window Processing" at the end of this chapter.

Retaining Window Information

Three prompts on the Customize window let you retain window information.

- Protect help** Prevents users from editing help messages you set up for the window. Otherwise, users can edit help messages that appear after they press [F1] (Context) or [Ctrl-F2] (Concept). The default entry is no help protection.
- Preserve keys** Specifies which parts of a multipart record key to preserve when the next record is displayed. This is for use *only* with multiple part keys. You can set up any multipart key prompt to preserve and display whatever data value was entered at this prompt when the previous record was displayed. The value will be displayed at the entry line from then on for each new record until the entry is changed. This eliminates the necessity to re-enter unchanged values.

To retain the value entered at a multipart key prompt, type the prompt order number of the multipart key prompt. More than one part of the multipart key can be specified. Enter each prompt order number on a separate line.

Preserve fields Specifies that the value entered at a prompt be retained for several records in succession. The current value will display at the entry line from then on for each new record until the user changes it. This eliminates the necessity to reenter unchanged values. For example, you might preserve a *State* prompt so that a data entry person does not need to retype the state abbreviation each time a new record is displayed. The cursor still moves to that entry line so the state can be changed if needed.

Use the *Preserve fields* prompt to specify which prompt values to preserve when the next record is displayed. Type the prompt order numbers of the prompts you want preserved. Enter each prompt order number on a separate line.

Adding Two Kinds of Window Help

Two kinds of help are available for windows. **Concept help** provides information about how to use the entire window. **Detail help** provides information on using the window prompts. Both kinds are specified in the Customize window with a code and command sequence. The Detail help process defined at the Customize window is called *only* at prompts that have not been assigned a prompt-specific Detail help code and command in the Prompt window. For more information on codes and commands, see the “Using Codes and Commands” chapter.

Concept help Indicates which kind of help to display when the user presses [Ctrl-F2]. Concept help should provide tips on using the entire window. Press [F2] (Options) at the *Concept Code* prompt in the Customize window for a list of available help codes and sample commands.

Detail help Indicates which kind of help to display when the user presses [F1] (Context). Detail help should provide information on using window prompts. For a list of available help codes and sample commands, press [F2] at the *Detail help Code* prompt in the Customize window.

The default detail help process specified in the Customize window is only called if no prompt-specific help is defined in the Prompt window. For more information about prompt-specific help see “Defining Prompt Processing” earlier in this chapter.

Creating Custom Softkeys

You can create a set of custom softkeys for the window you are defining. These softkey options will appear on the softkey popup for that window when the application runs.

Each softkey performs a unique function. For example, you might define a softkey option that displays additional information about a window or prompt, or one that displays a list of record keys.

Displaying the Custom Softkey Window

Use the Softkeys option to define softkey functions for your application window. Choose the Softkeys option from the Paint Main Menu to display the Softkeys window.

Each of the prompts on this window is described below.

Choosing a Key and Description

You specify which combination of keys will start a particular action at the *Key* prompt and include a short description of that action at the *User description* prompt. Press [F2] (Options) at the *Key* prompt to display a popup list of available keys. To choose a key, move the cursor to the key name and press [Enter].

Your user description should summarize the action performed by the softkey. It will appear in the Softkey popup.

It is a good idea to assign keys using logical groups of functions. For example, you might use Shift key combinations for all softkeys that display windows, and Alt key combinations for all softkeys that print reports.

Specifying Softkey Actions

Before a key will start an action, you must define that action by using a code and command sequence. For example, you might use the M code with a command that identifies a menu record to specify that the softkey will call a custom menu. For a list of available codes and commands, press [F2] from the code field. For more information on codes and commands, see the "Using Codes and Commands" chapter.

Relating Windows

Several windows can be displayed at a time if they are related. With related windows, users do not need to leave the current window to look up or modify information in another window.

You can define two different kinds of related windows using the Relations window. One type lets you view all the records for the related window. The other uses the relational indexing defined for a field in the other window to select a group of records to display in the related window.

When a related window has been defined for a window, the message “Relations” is displayed in the status line at the bottom of the screen. The user can display a related window by pressing the key combination assigned to that window. The Relations option is available on the [F6] Softkey popup or by pressing [Ctrl-F6].

Defining Related Windows

Choose the **Relations** option from the Paint Main menu to display the window.

To define a new related window, add a row of information for that window beneath the prompts in the Relations window.

Key	Indicates which key combination will display the related window. Press [F2] (Options) for a popup list of available keys. For consistency, try to use the same type of key combinations for all your related windows. For example, you might use various Alt key combinations to call each of the related windows in an application.
Source File	Specifies the file where the related window template is stored.
Related Window	Specifies the name of the related window template. The name should appear exactly as it was defined when painted.
Joined Field	Indicates the name of the field in the current window that contains keys to the records used in the related window. This field may be maintained through the use of a relational index. For more information on relational indexing, see the “Creating and Maintaining Indexes” chapter.

Defining Related Window Processing

To see the processing options for related windows, press the [→] key to display the second page of the Relations window.

By specifying codes and commands, you can perform processing before and after the related window is displayed. For instance, if you have restricted write privileges in the related window, you might run a process before the related window is displayed in order to display a message saying “Data displayed for information only, no updates can be made through this window.”

Press [F2] (Options) for a list of available codes. For more information on using codes and commands with related windows, see the “Using Codes and Commands” chapter.



Displaying a List of Related Windows

When an application is running, you can see a popup list of all the windows related to the current window by pressing [Ctrl-F6] (Relations).

Setting Window Video Attributes

Video attributes can be specified for any window. You can change attributes for the border, title, background, prompts, labels, and entry lines. The window's video attributes will override the video attributes you set up for a user or account.



Changing Window Video Attributes

Use the Color Override window to change the video attributes for the window you are currently creating. Choose the **Set Video** option from the More Paint Functions menu to display the Color Override window.

To change a setting, move the cursor to the appropriate prompt and press [F2] (Options). Choose the appropriate attribute from the popup list. The codes that indicate your choices will appear in the Set Video window just as the attributes will appear on the application window. For example, if you specify blink as an attribute for the window title, the code B will blink on and off in the Set Video window.

Establishing Window Security

Windows can be protected through accounts, a restriction level, and passwords. The restriction level specified for a user determines how window security will behave.

Window security interacts closely with user security. A user's restriction level determines whether a password is required for access to a window. If a user's restriction level is the same or less than the window's level, the user is given access without a password. If a user's restriction level is more than the window's level, they are denied access to the window, or a password is requested. For more information on the relationship between user and window security, see "Restricting Access to Windows" in the "Users and Accounts: Planning Access and Security" chapter.



Defining Window Security

Windows are secured using the Window Security window. To display this window, choose the **Security** option from the More Paint Functions menu. Descriptions of each prompt on the window follow.

- Accounts** Specifies which accounts this window will run in. You can specify more than one account. For example, you might have a Customer window that runs on both a Receivables account and a Sales account. Press [F2] (Options) for a list of all the accounts on the system.
- Restriction Level** Indicates the security level for this window to be compared with your restriction level. Level zero (0) provides the most restriction and level five (5) the least. For example, if a window has a restriction level of four, a user with a level of zero to four can access the window without a password. Only one restriction level is allowed for a window.
- Passwords** Specifies what you must type to gain access to a window if your restriction level is higher than that of the window. More than one password can be specified for a window.

Security Features

Security is checked whenever a user attempts either to run (Access) or to modify (Paint) a window. This is of concern particularly to developers who will be distributing applications using a full version of Advanced Revelation, rather than just a Run-time version.

If you have established security for a window, Paint will perform a security check before allowing a user to edit the window template. If a window security level has been established, the user must meet or exceed this security level before entering Paint.

If a password has been specified for the window, a user with an insufficient security level can use Paint to edit the window, as long as the user types in the proper password.

A brief summary of the security checks used by Paint:

- No password or security established: all users can Paint the window
 - Password (only) established: users who know the password can Paint the window
 - Security level established: users who meet or exceed the security level can Paint without typing in a password
 - Security level and Password established: users who match or surpass the security level or who type in the proper password can Paint the window
- ✎ The SYSPROG account is normally used by system administrators to establish and maintain security for other accounts. Because of this, security is ignored in the SYSPROG account. In order for any security to be in effect, an application must be running in a user (non-SYSPROG) account. Advanced Revelation applications that are sensitive to security should not permit users to access the SYSPROG account.

Disabling Window Keys

You can disable some keys available to users when an application window runs, to keep them from performing certain tasks. For example, a display-only window could have the [F9] (Save) key disabled to prevent updates to the file.

Displaying a List of Window Keys

To disable keys, choose the Key Disable option from the More Paint Functions menu to display a popup list of keys.

The [Enter] key acts as a toggle to disable and enable the keys appearing on the list. To choose a key to disable, move the cursor to the row for that key and press [Enter]. The message “disabled ” appears in the Disabled column. A disabled key is returned to its normal function by moving the cursor to the row for that key and pressing [Enter]. Choose as many keys as you would like to disable and press [F9] (Save) when finished to save your selections.

Press the [PgDn] key from the first page of this popup to display the second page.

Defining Collector Windows

Collector windows in Advanced Revelation function much like application windows, but do not write data to file. Instead, they pass the data they have gathered on to or back to some other process.

Many command windows in Advanced Revelation are collector windows. For instance, the RECORDCOPY window is a collector that gathers information about the copy process, then constructs and executes the appropriate TCL-level RECORDCOPY command. It does not – nor does it need to – write any data to a file.

Collector windows are intended primarily for application developers and require some understanding of R/BASIC syntax. To augment the information presented below, see the “Programming with R/BASIC” chapter in the *R/BASIC* section.

Processing @RECORD

Both types of windows process data using the variable @RECORD. While a record is active in a window, it is being held in @RECORD.

In an application window, different prompts on the screen are bound to fields in the dictionary of the data file. Data is read and written from and to @RECORD on the basis of the field bound to a prompt. When the user presses [F9] in an application window, the data in @RECORD is written to the data file.

A collector window is simpler. Rather than being bound through a dictionary, prompts in a collector window are simply bound to a specific position. The position refers to the element of @RECORD that will be updated by that prompt.

Why use Collector Windows?

As noted, a collector window simply creates fields in @RECORD. There are many ways to do this; why not use a simpler way?

The advantage of collector windows is that they offer the full capability of windows to the process of creating @RECORD. This includes the definition of prompts for each field, with all the attendant error checking, help message, default, and popup capability available to each prompt. In addition, a collector can make use of codes and commands for pre-, post-, and perpetual processing. In general, a collector provides for the most controlled input of data available in Advanced Revelation.

Creating a Collector Window

As with application windows, collectors are created with Paint. To tell Paint that a collector window is being built, use the word COLLECTOR when prompted for a data file name at the *Window Data File* prompt.

From this point, prompts on the screen can be painted in the normal way. Rather than being in Autobind mode, though, the binding method in a collector will be Number. This indicates that prompts will be assigned to a position in @RECORD as they are built. There will be no reference to a field name in the Prompt window ([Shift-F6]).

Most functions of Paint are available to collector windows. Each prompt can establish defaults and validation patterns, and the normal help, option and pre- and post-processing capabilities can be used. Multivalued and text prompts can be created. Keep in mind, however, that some of the standard Paint features, such as indexing, do not apply because Collector windows do not interact directly with files in the database.

Accessing a Collector Window

Under almost all circumstances, a collector window will be accessed by calling it from within an R/BASIC program. This is most easily done by using a CATALYST "W" call:

```
CALL CATALYST( "W", "COLLECTOR.NAME" )
```

The collector window displays and permits entry or edit of data in @RECORD. When the collector is exited, control returns to the calling program or process.

Sharing Data with Collectors (@PSEUDO)

Since the purpose of a collector is to manipulate data in @RECORD, the calling process must have a method to pass the data to and from a collector window. This is done using the global system variable @PSEUDO.

Whenever a window is opened on the screen, a new "window level" is established. This new window level has a complete new set of window variables (such as those maintained in AREV.COMMON). The new window also has a new copy of @RECORD. When the new window is closed, control reverts to the previous window level, and the original values for AREV.COMMON and @RECORD are again established.

Because the variable @PSEUDO is global across window levels, it can be initialized at one level, and read at another. This makes it possible to pass data between windows. A typical procedure is for a program to initialize @PSEUDO, and then call a collector via CATALYST. The collector window is able to read the value placed into @PSEUDO, and process it locally as @RECORD. The collector then updates @PSEUDO with the changes made in the collector, and returns to the previous level. The calling program can then access the changes by again looking at @PSEUDO.

A simple example of code that does this is illustrated in the following fragment:

```

READ @RECORD FROM CUSTOMER_FILE, @ID THEN
  @PSEUDO = @RECORD
  CALL CATALYST( "W", "TEST_COLLECTOR" )
  @RECORD = @PSEUDO
END

```

Collector Processing

A special option called **Collector** on the More Paint Functions menu is used to specify options unique to collector windows. This option addresses *Setup*, *Save* and *Exit* options.

Setup Options Setup options refer to processing that is to take place before the collector window processes the first prompt. There are two aspects to setup options: type, and code and command.

Type can contain a special code indicating what is to happen with the data passed in @PSEUDO. The four codes (0-3) are defined as follows:

Code	Meaning
0	nothing is done with @PSEUDO; data is not being passed
1	@PSEUDO is copied directly to @RECORD
2	@PSEUDO is copied to @RECORD, with @FMs and @VMs converted to spaces. This permits the entire contents of @RECORD to be passed into a single prompt in the collector.

Code	Meaning
3	@PSEUDO is copied to @RECORD, with @VM and @SVM converted to field marks. This permits a multivalued field to be passed to a collector, with each value assigned to a separate prompt.

In addition to the Type codes, a code and command sequence can be specified. A typical application would be to call a setup subroutine.

☛ The code and command sequence is executed *before* the type code is processed.

Save Options Save options are executed when a user presses [F9] at a collector window. The options available at this prompt are essentially the same as those at Setup, but work in the other direction. In other words, @PSEUDO is updated out of @RECORD.

For example, a Type of 2 will pass @RECORD to @PSEUDO while converting @FMs and @VMs to spaces. This would be used to create a single field out of the different prompts in the collector.

As with Setup, a code and command can be specified. These, too, are processed before the type code.

Exit The Exit option determines what is to happen to the window (as distinct from the data) when [F9] is pressed. There are three options:

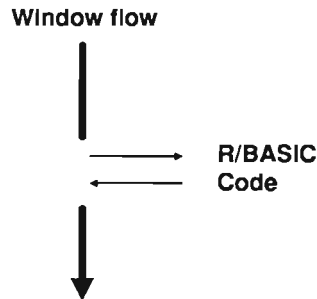
Exit Code	Meaning
0	The collector window closes, and returns to the calling process. This is used if the collector is a “one-shot” process.
1	The collector refreshes with no data in it. This creates an “anchored” collector.
2	The collector refreshes with the first prompt filled in with its previous value. This is used in system collectors where the file name for a process appears first.

For more information on codes and commands, see the “Using Codes and Commands” chapter. For more information on R/BASIC variables and AREV.COMMON, see the “Programming with R/BASIC” chapter in the *R/BASIC* section.

Adding Window Processing

Special processing on windows and prompts allows your application to process information automatically as it runs. Processing can be added to the window flow as needed with code and command sequences.

When a window runs, Advanced Revelation lets you call R/BASIC code at selected points to perform special operations. When the process is finished, control returns to the window.

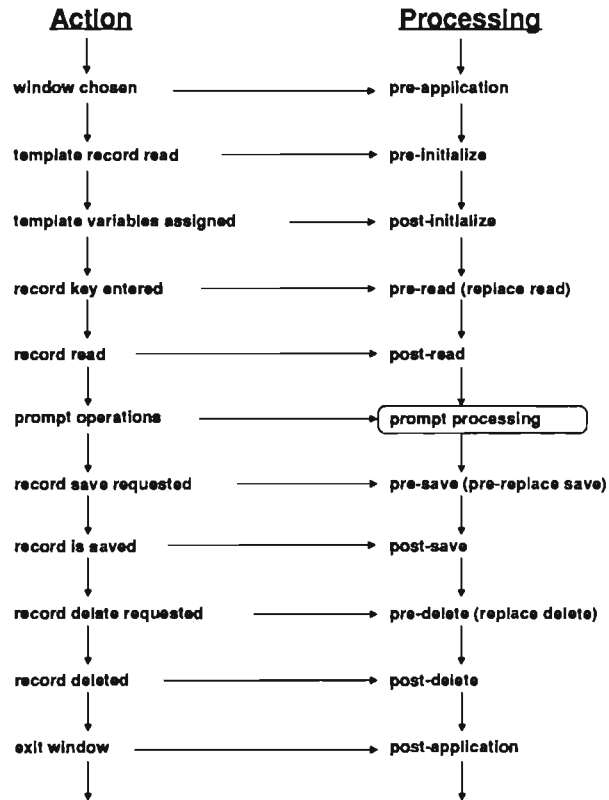


To use window processing, you need to understand what system variables are available at any given point in the window flow and what effect your code will have on those variables.

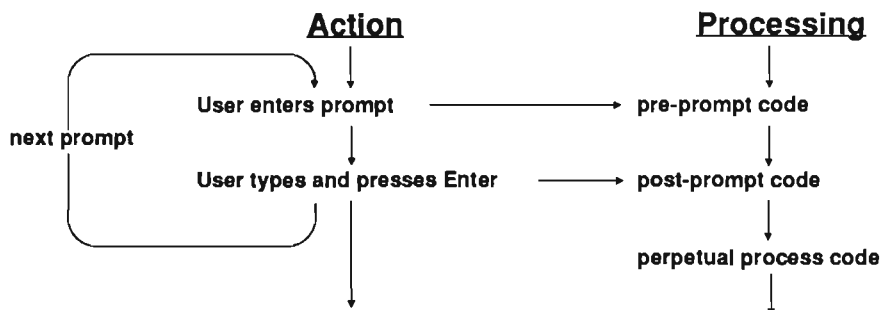
Descriptions of the exact points at which you can call a process and suggestions for using processing appear below.

Overview of Processing and Window Flow

Each type of processing takes effect at a different point in the window flow. The following illustration shows when each kind of window processing starts and stops:



The following illustration shows the flow of prompt processing:



Defining Window Processing

To define special processing for a window, choose the Processes option from the Paint Main menu. This displays the Process window.

Each prompt in this window is described below in the order in which it is executed. Descriptions of the pre-application and prompt processes are also included in the order in which they are executed.

The codes specified for each of the prompts are commonly S (subroutine) and { (dictionary record). The AREV.COMMON common variable area can be used to integrate subroutines and dictionaries that are called by code and command. Inserting the AREV.COMMON variables into the routines permits the sharing of these variables with the window processor. The system variable @ANS and the STATUS() function are used to pass information between the window and external programs.

- | | |
|------------------------|--|
| Pre-Application | Processes information before the window template is read. You might use a pre-application process to specify Table mode for a window and call an introductory help message. A pre-application process is specified on the command line when you start the window. For more information, see the WINDOW entry in the "TCL Command Reference" chapter in the <i>TCL</i> section. |
| Pre-Initialize | Processes information after the template record is read but before it is divided into variables and displayed on the screen. A process here might check to see what account this window is called from and remove several unnecessary prompts. |
| Post-Initialize | Processes information after the template record has been assigned to variables but before it is displayed on the screen. |
| Pre-Read | Processes information after the user has typed a record key and pressed [Enter]. It allows you to perform a process before the specified record is read. For example, you might have contractors and permanent staff on the same personnel list. You could check to see if that record is for a contractor and, if so, display a message window for contractors. |
| Replace-Read | Replaces the standard read procedure used by Advanced Revelation with your own read process. For example, you may want to use a special operating system read process that accesses parts of a record larger than 64K. |

- Post-Read** Processes information after the record is read, but not yet displayed. For example, you might have three fields called 30, 60, and 90 in a Receivables file that are used to display a message indicating when a customer is overdue. Your process could check a value in a particular field to determine whether to display one of these messages.
- Pre-Prompt** Processes information before the user starts typing information in a prompt. For example, you might do some special processing to see if you should skip this prompt. Pre-prompt processing is available for every prompt in the application window.
- Post-Prompt** Performs a process based on what has just been entered in the prompt. For example, you might call an R/BASIC subroutine in a phone number prompt that adds an area code to the number automatically if none was entered. Post-prompt processing is available for every prompt in the application window.
- Invalid** Performs a process in response to data that does not pass the In Pattern check. For example, if you typed in an invalid product type, the process could ask you if you want to add that to the valid product types.
- Perpetual Process** Performs a process between every prompt in a window. This process occurs immediately after the Post-Prompt process. For example, you might compare a payment amount with an invoice amount and, for all prompts where the payment is less than the invoice amount, change the display to the color red.
- Pre-Save** Processes information after the user presses [F9] but before the record has disappeared from the screen. For example, you might require that a minimum set of information be entered in a customer window (company, contact, and phone) before executing the save.
- Replace Save** Replaces the standard save procedure used by Advanced Revelation. For example, you could use a save process that writes to a specific device outside an Advanced Revelation file.
- Post-Save** Processes information after the record is written. For example, you might use a process to read each product key from a multivalued field in a product file. You could then subtract the number of products from the total quantity field to keep a running total of the quantity on hand for any item.

- Pre-Delete** Processes information after the user presses [Alt-D] but before the record is permanently removed. For example, you could write an invoice record to an invoice archive file before it is removed permanently. This would let you systematically archive records.
- Replace-Delete** Replaces the standard delete processes used by Advanced Revelation. For example, you could write a delete flag to the last field in a record and then use a process later to actually delete the record.
- Post-Delete** Processes information after a record is deleted but before the record key is removed from memory. For example, you could concatenate the record key and write it to a register. A post-application process could then save a list of all the keys written to that register.
- Post-Application** Processes information when the user exits the window. For example, you could write a key list of new or changed records to the LISTS file.

Using Registers in Advanced Revelation

Advanced Revelation windows make a special set of variables available to the developer. These variables are called registers. While registers may be initialized in Paint, they are only available when the window runs by way of subroutines.

Each window has a set of ten Registers, which can be initialized in the Customize window. Each prompt on a window has a set of five registers, which can be initialized in the Prompt window. Window registers are accessible through the AREV.COMMON variable REGISTER(*n*). (The *n* is the number for register 1 through 10.) Prompt registers are available through the prompt definition stored in the AREV.COMMON variable W(*n*) (where *n* is the number for register 1 through 5). Or, for the current prompt, the register is available through the dynamic array SI. The layout of SI may be found in the Labelled COMMON area LCPOSITIONS.

If a register is initialized in Paint, it should not contain Record, Field, or Value marks. If a register is initialized or used within a program, it is the responsibility of the developer to save and restore any values already in the register.

You must supply your own subroutines for manipulating registers. To use a window register with a code and command, use the following syntax at the *Command* prompt:

<R*n*>

The *n* is the number for register 1 through 10.

The following example uses the code C to compile and run the R/BASIC program in window register one. The code and command are entered at the Post-process *Save record* prompt in the Process window. The code and command:

Code	Command
C	<R1>

The R/BASIC code in window register 1:

```
IF {PAYMENT} GT 1000 THEN
  OPEN 'PAY_AMT' TO PAYFILE THEN
    AUDIT.RECORD = {PAYMENT} : @FM : @USERNAME
    WRITE AUDIT.RECORD TO PAYFILE, @ID
  END
END
```

The program checks the PAYMENT field and opens the PAY_AMT file if the payment is greater than 1000. An audit record is constructed and written to the file with the payment amount and the current user name.

For more information on codes and commands, see the “Using Codes and Commands” chapter. For more information on R/BASIC variables, see the “Programming with R/BASIC” chapter in the *R/BASIC* section.

Developer Notes for Paint

This section provides information about the ways you can customize the processes associated with the [Alt-V] key in a window. The section also provides information about how the window processor handles recalculation of symbolic prompts in a window.

Customizing the (Alt-V) Process

Developers can customize the function associated with the [Alt-V] key in a window. Normally, this key calls an R/LIST window that enables a user to construct and execute an R/LIST command. If a browse list is active at the time the user presses [Alt-V], the browse list will be converted to a select list before the R/LIST command is executed.

Settings in both Paint and in the Environment window enable you to customize the contents and appearance of the R/LIST window. These customization options permit you to specify a default R/LIST sentence that is to appear each time a user presses [Alt-V], or to substitute an alternate window or process for the default R/LIST window.

- ☛ Use of the [Alt-V] key is discussed in the “Creating Reports with Advanced Revelation” chapter in the *Creating Reports* section.

Creating a Default R/LIST Template

The *View R /LIST template* prompt in the Customize window of Paint enables you to specify a default R/LIST sentence that appears each time [Alt-V] is pressed. This saves users the trouble of having to type in a full R/LIST sentence in the R/LIST window each time they press [Alt-V].

Enter at this prompt the name of an R/LIST template from the QUERIES file. You can create an R/LIST template in the QUERIES file by using the **Design-Report-Rlist** menu options.

For more information on using the Make an Rlist Report window, see the “Creating R/LIST Reports” chapter in *Creating Reports*.

If you specify a default template in Paint, this R/LIST template will display in the R/LIST window whenever the [Alt-V] key is pressed in this window. You can edit this R/LIST command each time it appears in the R/LIST window.

Alternatives to the R/LIST Window

If you do not fill in the *View R /LIST template* prompt in the Customize window, you can specify an alternative to the R/LIST window. This permits you to bypass the R/LIST window altogether, and substitute your own window or process for the R/LIST window. Your process will be called each time the [Alt-V] key is pressed in that window.

Paint You can customize the [Alt-V] key by filling two prompts in the Customize window: *View R /LIST builder code* and *View R /LIST builder command*. These prompts accept a code and command used to substitute a specified process for the R/LIST window.

General System Configuration Window

Even if you do not fill in these prompts while using Paint with any specific window, you can create a default substitution for the R/LIST window. To do so, enter a code and command at the *View Mode Sentence Builder* prompt in the General System Configuration window. Use the **Management-Environment-General** options from the Advanced Revelation Main menu to access the General System Configuration window.

The code and command filled in at the *View Mode Sentence Builder* prompt in the General System Configuration window is also used by View. For more information on the use of this prompt by View, see “Developer Notes for View” in the “Creating Reports with Advanced Revelation” chapter in *Creating Reports*.

- ☛ Use of a View R/LIST builder code and command in the Customize window in Paint takes precedence over the

View mode sentence builder specification in the General System Configuration window.

In either case, the code and command process will be executed instead of the R/LIST window. For instance, you might specify with a “W” code that [Alt-V] is to display a window that you have created, or you might specify with an “S” code that [Alt-V] is to call an R/BASIC subroutine that you have written.

If you wish to have the substitute window execute an R/LIST report using the browse list as a select list, you should load the appropriate R/LIST command into the system variable @PSEUDO. When your process has finished, the window will execute whatever is in @PSEUDO, using the browse list as a select list. If @PSEUDO remains null, the original application window will be redisplayed with no further processing.

Recalculating Symbolic Fields Based on Dependency

The Paint Recalculate option that updates symbolic fields dependent on the current prompt creates and maintains two pieces of control information. The first is a dependency table in the window template itself, and the second is a special record in the data dictionary that tracks templates dependent on the file.

Dependency Table in the Template

When you choose the dependent prompt option for symbolic field recalculation, Paint builds a dependency table in position 34 of the window template. When the window is running, this table is available in the AREV.COMMON variable called REC.MAP.

Creation of Dependency Table

Paint builds the dependency table by examining the source code of the formulas for symbolic fields. Dependencies are defined as references to other fields within the formula. The references can take two forms: references within braces (for example, {NAME}), or references to an @RECORD extract in which the position is hard coded (for example, @RECORD<1>).

For example, this formula contains two dependencies:

```
{SUBTOTAL} + @RECORD<3>
```

When the dependency table is created, the field using this formula will be dependent on any updates to either SUBTOTAL or field 3 of the current record.

Structure of Dependency Table

The dependency table is a multivalued field, with one value for each prompt in the window. In each value is a list

(comma-delimited) of the prompts in the window dependent on that prompt.

For example, a dependency table might look like this:

$$\begin{matrix} & 2 & 4 & 5 & 2 & 2 & 2 \\ 2 & 4 & 5 & 2 & 2 & 2 & 2 \end{matrix}$$

This table is for a window containing six prompts. Prompts 4 and 5 are symbolic fields. The field name for prompt 2 is referenced in the formulas for prompts 4 and 5. The field name for prompt 3 is referenced in prompt 5.

With this table in place, the window processor can make a decision about the prompts to update if any data in the window changes. For this example, if the user changes prompt 2, the window knows to update the symbolic fields in prompts 4 and 5.

You can change the dependency table after it has been created by Paint. For example, you might add the prompt numbers of fields to be recalculated that are not necessarily dependent on specific fields.

- ☛ Paint rebuilds the dependency table *every* time the window template is saved. If you use the full-screen Editor to alter the window template dependency table, your changes will be lost if you subsequently access and save the window within Paint.

When a symbolic field definition is edited in the dictionary, DICT.MFS can update only the dependency tables of templates that are attached at the time of the edit. To ensure that the appropriate dependency tables are updated when a symbolic field definition is changed, you can attach all templates that access and update the data file containing that symbolic field, prior to editing. To update the dependency table of a template that was not attached when one of its symbolic fields was edited, redisplay that template in Paint and save it.

Using Paint to Customize Query

Several processing hooks have been incorporated into Paint to accommodate Query.

Window Flow in Query

Query operates by dynamically creating a collector window based on the specifications for the original data entry window. This collector window is then displayed by recursively calling the window processor.

When customizing the Query process, it is important to be aware of which window (application or collector) is being processed. For example, the Pre-Query process can access the

AREV.COMMON block, but the values in the common block are applicable to the collector window, not to those in the original data entry window.

Query/Index Popup Code and Command

Use the *Query/Index popup* code and command prompts in the Customize window to specify what is to happen when Query returns a select list. This permits you to modify or refine (but not clear) the select list. For example, your code and command might call a multiselect popup from which the user can edit the list returned by the Query process.

If you do not specify a *Query/Index popup* code and command, the list will be returned intact to the original data entry window as a browse list.

- ☛ The list returned to the window is limited to approximately 32K in length. If the select list generated by a query is very long, it will be truncated to this length.

At the time your routine is called, any records that were selected will be in an active select list. Your routine can use this select list as you require without affecting the list to be passed to the application window. For example, if your code and command performs a report, the report will work against the active select list. The active select list is preserved, however, and is returned to the window as a browse list (see “Post-Query Processing” below for information on clearing this list).

If you wish to modify the selected list of record keys directly, you can load the variable @ANS. If @ANS contains any information (in other words, is not null) at the time your routine finishes, the data in @ANS will become the browse list in the application window. Data in @ANS should be delimited with a value mark. If @ANS returns from your process with nothing in it (in other words, is set to null), the original select list will be used as the browse list.

Pre-Query Processing

You can create a Pre-Query process to determine what is to happen before a query is executed. Use the Processes window in Paint to enter a code and command that defines the Pre-Query process.

The Pre-Query process gives you an opportunity to manipulate the query generated by Query before it is executed. The Pre-Query process is called when the user presses [F9] at the Query window, or [F9] after editing the command in the Query Select window, but before the query command is executed.

When the Pre-Query process is called, the SELECT command generated by the Query process has been loaded into the AREV.COMMON variable RLIST.SELECT. If you change this variable, you will modify the command executed by the Query process. You also have access to the raw query information (the literal information the user has entered), since this information is stored in @RECORD.

The Pre-Query process is called as a part of the Query collector window (as a post-process to that window). Access to @RECORD and to AREV.COMMON from the Pre-Query process gives you the versions of these variables that are active at the time the Query collector window is active.

Post-Query Processing

You can also create a Post-Query process to be called after the query has been executed. Use the Process window in Paint to enter the code and command defining the Post-Query process.

The Post-Query process enables you to manipulate the results of a query after the record keys have been loaded into the browse list, but before the application window regains control. This permits you to refine or clear the results of the query.

At the time the Post-Query process is called, certain AREV.COMMON variables have been set. These are:

Variable	Contains
BROWSE.NEXT	Results of query (up to approx. 32K)
NEW.BROWSE	1 (meaning a new browse list is active)
RESET	5 (to cause a window refresh)

The Post-Query process can modify any or all of these. To clear the browse list completely (for instance, if the Query/Index Popup process has already processed the records), set BROWSE.NEXT to null.

Unlike the Pre-Query process, the Post-Query process is called out of the application window itself. Any changes you make to @RECORD or to AREV.COMMON will have an affect on the application window directly.



Chapter F 6

Building Menus and Popups

You can create menus and popups to use with your application windows. This chapter explains how to plan a menu system and define menus. It also explains how to design and create popups.

Designing and Building Menus	F 6 . 1
Displaying a Main Menu in an Application	F 6 . 3
Designing and Building Popups	F 6 . 4
Displaying Popups with Codes and Commands	F 6 . 13



Designing and Building Menus

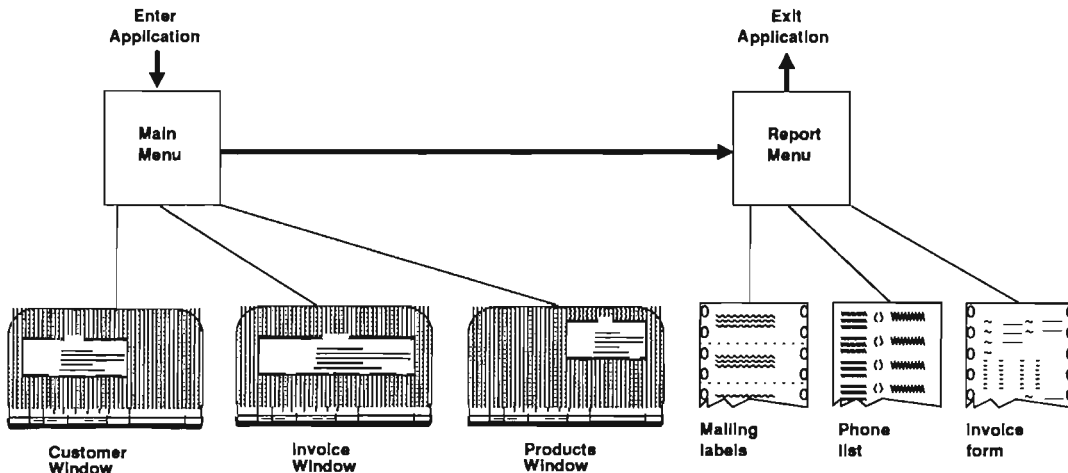
Menus give users access to application windows and reports without having to learn the Advanced Revelation commands and menu system. By choosing options from menus, users can run application windows, print reports, and a variety of other tasks.

You can create different menus for different users. For example, a data entry person may be given a different set of options than an accountant in a payroll application.

Planning a Menu System

Design the menus for your application after you have set up the application windows and reports. That way you can be sure to include a menu option for all the application parts.

Group your menu options together according to order and frequency of use. For example, an application might group all its window options on a main menu and all the report options on a report menu accessible from the main menu. It might also provide a way to exit the application from the report menu.



Note that menus can call other menus. For larger applications, you may want to use a number of different menus all accessible from one main menu.

Using the Make Menu Window

To build a menu, choose the **Design-Menu** options from the Advanced Revelation Main menu. The Make Menu window displays.

Enter each menu option specification as a separate row beneath the *Selection*, *Description*, *Code*, and *Command* prompts. When the menu is run, the options will appear on a menu that looks much like the Advanced Revelation system menus.

Descriptions of each prompt in this window follow. When you are finished defining your menu, press [F9] (Save).

Menu Name	Assigns a name to this menu. Application menus are stored in the MENUS file. For list of menus already defined, press [F2] (Options).
Title	Specifies the name that appears on the menu. Note that the length of the title is limited by the width of your computer screen.
Selection	Indicates the word(s) that will display as the menu option. Menu options appear in the order they are entered in the Make Menu window.
Description	Indicates the information that will appear at the bottom of the menu when a menu option is highlighted.
Code and Command	Specifies the action to take when a menu option is chosen. Codes commonly used on menus include: • W for running an application window • M for displaying another menu • P for running a popup window • E for executing a TCL command For more information on codes and commands, see the “Using Codes and Commands” chapter in <i>Developing an Application</i>.
Columns Across	Determines how many menu options will appear in each row of the menu. For example, an entry of 4 indicates 4 options per row.
Anchor	Specifies whether the system will return to this menu after carrying out the action specified for a menu option. For example, if you specified Y for yes on a menu that displays application windows, the system would redisplay that menu when the user left a window.

Displaying a Main Menu in an Application

When a user starts an application, Advanced Revelation can display the main menu built for your application instead of the Advanced Revelation main system menu. Advanced Revelation can also display different menus for different users.

The display of opening menus for an application is determined by a prompt in the General System Configuration window for a particular user or account, or by a Restart process specified at the User and Account Security windows. For more information on environments and security, see the “Users and Accounts: Planning Access and Security” chapter in *Developing an Application*.

Specifying a Menu Using an Environment Record

To specify the first menu to display when a user logs on to Advanced Revelation, you can change the environment record for the user or account. To do so, log on to the SYSPROG account (AREV SYSPROG). Then display the General System Configuration window by choosing the **Management-Environment-General** options from the Advanced Revelation Main menu.

To choose which environment record to edit, press [Shift-F1] from the General System Configuration window. This displays a popup list of all the environments you have created for different users and accounts. Position the cursor on the appropriate environment name and press [Enter]. The settings for that environment display in the window.

The *Default system mode* prompt determines whether a menu or the TCL or SQL window is displayed when a user logs on to Advanced Revelation. These are the options for this prompt:

- M Displays the default System menu
- T Displays the TCL window
- S Displays the SQL window

If you have specified M, the *Default system menu* prompt determines which menu is displayed. To display the first menu in your application when a user logs on to Advanced Revelation, change the entry at the *Default system menu* prompt from MAIN to the name of the first menu for your application. When you are finished, press [F9] (Save) to save your entry.

Displaying a Menu Using the Restart Process

When you define user and account security, you can specify which menu to display first with a Restart code and command. A Restart process starts a command when a user logs on to Advanced Revelation. To specify a Restart code and command, use the Secure Account or Secure User window. To specify which menu displays, enter M at the *Restart Code* prompt. At the *Restart Command* prompt, enter a command of *menu.name*, where *menu.name* is the key to a record in the MENUS file. For

more information on the Restart process, see the “Users & Accounts: Planning Access and Security” chapter in *Developing an Application*.

Designing And Building Popups

Using popups, you can display information to the user in table form. In some popups, the user chooses one or more rows of information to use in further processing. In other popups, information is only displayed. In either case, the displayed information can be drawn from a file or can be explicitly written as part of the popup definition.

Popups can return the value or values selected by the user to the location the popup was called from (for example, to a prompt in an application window).

Each popup definition is stored as a record in a file. Once you define a popup with the Make Popup window, it can be called from any point in your application. You can assign the popup to be called with the [F2] Options key, a special softkey, or a menu option by using a popup code and command sequence.

Planning for Popups

Examine your application and try to anticipate places where users may need additional information. For example, you might have a popup available from a Contact Name prompt that displays the names of all the contacts that work for a particular company. Or, at a Product Code prompt, you might supply a popup that lists codes and descriptions to remind a user what products are normally carried.

Using the Make Popup Window

To design a popup, choose the **Design-Popup** options from the Advanced Revelation Main menu. The Make Popup window displays.

The Make Popup window has two pages of prompts. The use of these prompts is explained below. Make the appropriate entries and then press [F9] (Save) to write your popup definition to the specified file.

- ☛ To preview your popup while you are defining it, press [Shift-F1]. Advanced Revelation displays the popup as it would appear according to your current definition.

Choosing a Popup File and Name

Use the first two prompts in the Make Popup window to specify the file and record where your popup definition is stored.

File name Specifies the system file that stores the popup records. By default, Advanced Revelation stores your popup records in

the POPUPS file. To store the record in a different file, enter the file name here.

Record Name Specifies the name of the popup record you want to create or modify. The popup record is stored under this name in the file indicated at the *File name* prompt. The system locates the popup using the record name you give it. Press [F2] (Options) for a list of existing popups.

Determining the Popup's Location on the Screen

Use the *Column* and *Row* prompts to position your popup on the screen.

Column and Row Indicates the starting position for the upper left-hand corner of the popup. Consider the size and location of your popup in relation to other windows on the screen. For example, if an application window appears in the upper left-hand corner of the screen, you may want to specify 35 as your starting column and 20 as your starting row so that the user can see both the window and the popup at the same time.

Choosing a Selection Process

Each popup you define can return a single value or multiple values to the location the popup was called from. For example, you might want to return the name of only one contact to a contact name prompt. However, you may want to return the keys for several records to a key list prompt.

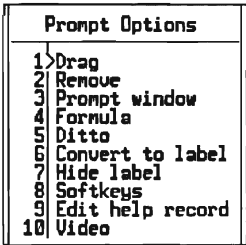
The *Selection process* prompt specifies how many values are returned by the popup and the characteristics of each type of multiple valued popup. These are the options for this prompt:

Option	Characteristics
NULL or 0	One selection
1	Multiple selections [F9] to accept
O	Multiple ordered selections

Descriptions and examples of each type of popup follow.

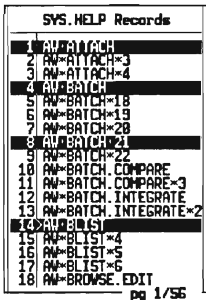
Single (NULL or 0)

A single selection popup returns one value. An example of a single selection popup is the following Prompt Options popup available in Paint.



Multiple (1)

On a multiple selection popup, the user indicates selections by positioning the cursor and pressing [Enter] for each option. The user presses [F9] (Save) to save the selections. An example of a multiple selection popup is the following list of record keys called from the Source records prompts on the Recordcopy window.



On this popup, the user can select several template records to copy by positioning the cursor on a record and pressing [Enter], positioning the cursor on another record and pressing [Enter], and so on. When the user presses [F9] (Save), the name of all the records to copy are returned to the *Source record(s)* prompt in the Recordcopy window.

You can also use multiselect popups at associated multi-valued prompts in windows.

For example, suppose you have an invoice entry window containing associated multivalued prompts for product codes and quantities. You can create a popup that will return more than one product code at a time. If you call this popup at the product code prompt (using the Options key, for example), you can then select a group of product codes, and have them

returned as a group back to the multivalued product code prompt.

Ordered (O)

An ordered popup lets you select multiple options in a specific order. For example, the following Quick Paint popup lets you choose fields from the CUSTOMER file to display in a window. The order in which items are selected indicates the order in which fields are assigned as prompts in the window you are painting. A number indicating the order appears next to each option.

Select fields from dictionary to become prompts in window				
	Field Name	Number	Type	Order
1	>ADDRESS	3	F	
2	CITY	4	F	
3	CITY.XREF		F	
4	COMPANY	1	F	
5	COMPANY.XREF		F	
6	CONTACT	2	F	
7	CONTACT.XREF		F	
8	CUST.NO	0	F	
9	INVOICES	9	F	
10	NOTES	8	F	
11	PHONE	7	F	
12	RECORD.ID	0	F	
13	ST	5	F	
14	ZIP	6	F	

Defining the Information the Popup Displays

The popup mode specifies the criteria used to retrieve the information that will be displayed in the popup. The displayed information can be drawn from a file (specified at the *Source file* prompt), or it can be explicitly written as part of the popup definition. You use the *Explicit popup information* prompt to specify which information is to be drawn from the source file, or to explicitly write the information you want displayed.

First, enter the popup mode you want at the *Mode* prompt.

Mode (determined by data characteristics)

The *Mode* prompt specifies the popup mode.

Mode:	Uses:
T (Table)	The information displayed in the popup will consist of multiple records read from a file. Each selected record from the file is displayed as a row in the popup, with each selected field becoming a column in the row.
R (Record)	The information displayed in the popup will either be read from a file or explicitly written as part of the popup definition. If the information is read from a file, a single selected record is displayed. Each selected field from the record is displayed as a row, with each selected value becoming a column in the row. If the information is written as part of the popup definition, you specify the information at the <i>Explicit information</i> prompt, explained later in this section.
F (Field)	The information displayed in the popup will be a single field from one record from a file. Each selected value from the field is displayed as a row, with each selected subvalue becoming a column in the row.

After determining the popup mode, you enter appropriate information at the *Source file* prompt and the *Explicit information* prompt, depending on the popup mode. Descriptions and examples of each mode follow, including explanations of the *Source file* prompt and *Explicit information* prompt for each mode.

Using Table Mode (T)

A popup using Table mode selects records either from a key list specified at the *Explicit information* prompt, or from a list it finds or develops from your system. Each record is read from the file and becomes a row in the popup, with the record's fields becoming columns.

Source File for Table Mode

Table mode requires an entry at the *Source file* prompt. Enter the name of the file that contains the records you want displayed.

Determining Which Records to Display

You can specify a key list at the *Explicit information* prompt, in which case each line in the list is a key to a record in the source file, and each record is displayed as a row in the popup.

If you do not specify a key list at the *Explicit information* prompt, then the popup uses one of three kinds of record key lists in the following order:

1. An active select list of record keys for the file.
2. The file's Quickdex sorted record key list (if one exists).
3. A list of all the record keys in the file, created when the popup is called.

If no key list is specified at the *Explicit information* prompt, the system first checks to see if a select list for that file is active. If it finds one, it retrieves the records specified. Otherwise, it looks for a Quickdex sorted list of record keys for the file. If it cannot find a select or Quickdex key list, the popup selects the entire file and uses all the record keys to display information.

For more information on selecting lists of records, see the "Selecting and Browsing Through Data" chapter in *Accessing Data*. For an explanation of Quickdex sorted key lists, see the "Creating and Maintaining Indexes" chapter in *Developing an Application*.

Using Record Mode

Using Record mode, you can specify a single record from a file to be displayed, or you can write the information to be displayed at the *Explicit information* prompt.

Source File for Record Mode

If the information displayed on the popup will be read from a file, enter the file name at the *Source file* prompt.

If you want to write the information to be displayed explicitly, leave the *Source file* prompt blank.

Determining Which Record to Display

If the information will be read from a file, a single selected record is displayed. Enter the key for the selected record at the *Explicit information* prompt. Each selected field from the record is displayed as a row, with each selected value becoming a column in the row.

If you want to write the information to be displayed explicitly, enter the information at the *Explicit information* prompt. Each new line you enter at the prompt is a separate row in the popup. Use the bar character "|" (a shifted backslash key) to separate columns in the rows.

For example, you might specify the following information to be displayed in your popup:

Explicit popup information (depending on mode)							
JB:John Bertrun;Pencon, Inc.							
KP:Kurt Prince;Kaiser, Hendendhall and Prince Group							
LT:Loni Taylor;Aupro Corporation							
=====							
Window	Enter	Mu			SoftKeys	POPUPS	New Record
Ed Ins		Row:	3/3		Sysprog		Level 2

This would display the initials, full name, and company name for three contacts. You might call this popup from a contact prompt in a customer window and return the initials in the first column to the window. For details on returning values from a popup, see “Specifying Which Popup Column to Return” later in this chapter.

Using Field Mode (F)

Using Field mode, you can specify a selected field from a record to be displayed in the popup.

Source File for Field Mode Field mode requires an entry at the *Source file* prompt. Enter the name of the file that contains the records you want displayed.

Determining Which Field to Display At the *Explicit information* prompt, you specify the record key and position number for the field you want displayed. Type in the key on line one and the position number on line two.

For example, to display the sixth field from the MASTER_PARAMETERS record in your source file, you would make this entry at the *Explicit information* prompt:

```
MASTER_PARAMETERS
6
```

Each value in the field is a separate row in the popup, with each subvalue becoming a column in the row.

Specifying How Popup Information Appears

Using prompts on the second page of the Make Popup window, you can specify how information for each item will appear on the popup. Press the [↓] or [PgDn] keys to display the second page.

Explanations of the prompts on the second page of the Make Popup window follow.

Title Specifies the title that appears at the top of the popup. Titles can be one or more lines long.

Column Formats Each row of format specifications under *Column Formats* indicates the characteristics for one column of information in the popup. A description of each prompt follows.

Position The way that the *Position* prompt is used depends on whether or not information was entered at the *Explicit information* prompt on the first page of the Make Popup window.

When the *Explicit information* prompt contains data, the *Position* prompt indicates the column to which the format applies. Each line of data entered at the *Explicit information* prompt is separated into columns by a bar character “|” (a shifted backslash). Use the *Position* prompt to indicate which column will use that format specification. 1 indicates the first column, 2 the second column, and so on. For example, you might specify the following information for the popup of contacts described in the previous section.

<i>Position</i>	<i>Width</i>	<i>Just</i>	<i>Column Heading</i>
1	8	L	Initials
2	12	L	Contact Name
3	15	L	Company

When Mode is:	Position refers to:
T (Table)	The position in the record
R (Record)	The value position in the field
F (Field)	The subvalue position in the value

Typically, if no *Explicit information* prompt entry is made, the popup will be used to display record keys from a specified file. This is accomplished by specifying T (Table) for *Mode*, the Source file name, and entering a zero (0) at the *Position* prompt.

Column Heading Specifies the name that appears at the top of the popup for this column of information. Avoid using the bar character (|) in column headings.

Width Allocates the specified amount of space for displaying information from this column on the popup.

Just Indicates the justification to use for displaying information from this column. Available justification options are:

- L Left
- R Right
- C Centered

Conversion Allows you to specify a data conversion option to perform on the information displayed in this column. Press [F2] (Options) for a list of available options. These options are the same as the ones available for the *Output format* prompt in the Prompt window in Paint. For more information on these options, see the section on “Data Formatting” in the “Adding Advanced Features to Windows with Paint” chapter in *Developing an Application*.

Specifying Help for Popup Options

To specify help information displayed when the user presses [F1] from a popup, use the *Popup detail help key* prompt. Enter the name of a record from either the HELP or SYS.HELP file to display as detail help for the popup. For information on creating popup help records, press [F1] (Context help) at the *Popup detail help key* prompt.

Specifying Which Popup Column to Return

A popup can display several columns of information but can only return one column of information to the process that called the popup. Use the *Column to Return* prompt to specify which column to return. An entry at the *Type of Information* prompt (explained below) overrides an entry at the *Column to Return* prompt.

The number you specify at the *Column to Return* prompt indicates which column of information to return from the popup. An entry of 1 indicates the first column, 2 the second, and so on.

When the *Explicit information* prompt contains data, each line of data is separated into columns by a bar character “|” (a shifted backslash). Use the *Column to Return* prompt to indicate which column will return data to the process that called the popup.

If the information to be displayed in the popup will be called from a file, the columns are numbered in the order that they appear in the file. For example, the columns in a Table mode popup would be numbered consecutively, starting with the first selected field from the record to be displayed.

Specifying a Return Type

The last prompt in the window determines what sort of information to return for each row on a popup, if any. An entry at this prompt will override an entry at the *Column to Return* prompt explained above.

These are the options for the *Type of information returned* prompt:

Type	Returns
K	Returns the record key for the row or rows of selected information
R	Does not return a value. It indicates that the information is only to be displayed on the popup and not returned.
P	Returns the row number of the item or items selected from a popup, not a value displayed in the popup.

Displaying Popups With Codes And Commands

You can define a call to a popup anywhere that a code and command sequence is permitted. Popups are commonly defined as options on individual application window prompts.

Using the P Code

Popups are called using the P code. The general format is:

`P   FILENAME*POPUPNAME`

FILENAME is the name of the file that stores all your popups and *POPUPNAME* is whatever name you have specified for that popup record. Separate the file name and record name by an asterisk (*).

For more information about the use of codes and commands, see the “Using Codes and Commands” chapter in this section.

Display Options for Popup Commands

Formatting options are available for popups that display system file names or record keys. These options affect popups that use one of the following as the command portion of the calling code and command:

Command	Meaning
@ID, @IDS	Return one or more record keys
@FILE, @FILES	Return one or more currently attached file names
@VOLUME, @VOLUMES	Return one or more currently attached volumes

Options for these popup commands enable you to specify the display width, and the column and row coordinates where the popup is to appear.

The syntax used in the command portion of the appropriate code and command is:

`@ID:display_width:column:row`

(Any of the commands listed above can substitute for the @ID portion of the example.) Do not put any spaces in the command.

For example, suppose you specify the popup code and command:

<i>Code</i>	P
<i>Command</i>	@FILES:45:15:5

When this popup is called (perhaps with the Options key at a window prompt), a popup will be produced that displays all the currently attached files. The user will be permitted to choose one or more files from this popup. The data portion of the popup will be 45 characters wide. The top left corner of the popup will appear in column 15, row 5 of the screen.

You can skip any of the three additional parameters for the command, but must leave a colon in place to mark the spot. For example, if you did not wish to indicate a display width for a popup, but did want to specify a column and row position, the command might read:

`@FILES::15:5`

The column width will simply revert to its default value. The default values for each of the three parameters are:

Parameter	Default Value
Display Width	25
Column	0
Row	0

Chapter F 7

Creating and Maintaining Indexes

This chapter describes Advanced Revelation's main indexing systems — Btree, Cross Reference and Relational — and how they can be used to enhance your application. It also describes the utilities provided to update and maintain Advanced Revelation datafile indexes. For information on using established indexes in application windows, see the “Using Indexes” chapter in the *Accessing Data* section.

Introduction	F 7 . 1
Creating Indexes	F 7 . 3
Using Quickdex	F 7 . 10
Indexing System Maintenance	F 7 . 13
Reviewing Index Specifications	F 7 . 16
Updating and Rebuilding Indexes	F 7 . 17
Developer Notes for Indexing	F 7 . 22



Introduction

Advanced Revelation offers many different techniques to locate records in your database. The primary access method, known as hashing, is the fastest technique ever developed to retrieve records. Hashed record retrieval does not degrade as the database grows — retrieval time for a record is just as fast for a database with 100 records as it is for one with 100,000 records.

Hashing also allows Advanced Revelation to be a truly networked database. It is not necessary to lock an entire file or index when a record is being updated. Most fixed length databases using indexes as their primary retrieval method cannot be networked since they must lock part or all of their index while updating.

Advanced Revelation contains secondary indexing facilities so that you can retrieve records from your database without knowing the key (for example, retrieving a customer by name, city, area code, etc.). If an indexed field is used in a sort or select, the processing time will be greatly reduced.

The Purpose of Indexing a Field

To index a file, you specify which fields in the record will be used to create the indexes. Thereafter, you can access data records quickly through these fields. An index takes a value that you provide and uses it to return the record key. That key is used by the system to find the record.

In an index, the keys to the records are maintained in a sorted order that is defined by the values in the indexed field. For record searches, when you supply a known value to the index, it returns the key or keys to records that contain that value.

For example, in a CUSTOMER file, you might index the ST (state) field. In this index, the keys to the records will be ordered by state. As each customer record is entered, the ST field index is updated.

Benefits of Indexing

Indexing provides you three different benefits. Indexes save processing time when you want to sort records for reports. A sort arranges the report according to the order of the contents of a specified field. When the sort is based on an indexed field, the R/LIST processor can use the existing index key list, which is already sorted, rather than passing through the entire data file to get the sort values. An index also saves time when you want to select records for reports or other processing. Your selection criteria are applied to the index, rather than to the whole data file, so Advanced Revelation has less searching to do. Finally, indexing is used to retrieve records of information from files in the database when the keys are not known. When you don't know the key to a record, indexing allows you to use information from a field in the record to obtain the record key.

Determining the Need for Indexing

Indexes take up extra disk storage space. To save disk space, index only the fields that are most useful for fast retrieval purposes. At the same time, however, defining indexes will save processing time when working with a very large data file.

When designing an application, consider the most common reports and how information should be sorted and selected for the reports. Ask yourself what is needed for quick update and retrieval of records. This will affect index design.

For example, in designing a CUSTOMER file that will usually contain 100,000 records, think about how you will present the database information on reports. Consider the most common selection criteria and how to present the records most logically. Perhaps the prevailing selection criteria will be based on the value of state, and the usual sort for reports will be by company. In this case, define indexing for the STATE and COMPANY fields.

Defining Indexes

You can define indexes in three places in the system. Use:

- [Shift-F1] Indexing softkey in the Dictionary window
- **Indexing** option on the More Paint Functions menu
- **Indexing** option on the File Tools menu.

If you are defining the record format in the Dictionary window prior to painting application screens, add indexes with the indexing softkey [Shift-F1]. If you are creating the record format while using Paint, use the **Indexing** option on the More Paint Functions menu. To add or update indexes in an existing file, choose **Indexing** from the File Tools menu.

- ☛ If using Advanced Revelation on a network, it is easiest to add indexes to an active data file (one containing data records) during a quiet time on the network. Once a new index is added, make sure that the file is immediately re-attached by all network stations that write to it. This insures that the indexing system is properly updated with records entered at other stations on the network.

If other work stations do not re-attach at this time (by using the ATTACH command or by logging on), and continue to write records, those records will not be indexed. Thus, the new index will be inaccurate. To correct this situation, you must rebuild the index using the **Update** option from the File Indexing menu. For more information about the **Update** option, see "Updating and Rebuilding Indexes" later in this chapter.

Creating Indexes

Advanced Revelation provides three types of indexing: Btree, Cross Reference, and Relational. The kind of indexing that you define depends on how it will be used. You can specify any or all types of field indexing for a field. For example, you can specify that a company name field be both Cross Reference and Btree indexed.. This section describes the features of the three types of indexes, and when and how to use them.

Btree Indexing

Btree indexing uses the contents of the entire field as an index key. For instance, in a Btree indexed company name field, the complete company name would constitute an index entry. This distinguishes it from a Cross Reference indexing, in which each word in the company name would become an entry in the index. When searching a Btree indexed field for records containing a particular value, either you must supply the exact contents of the field or you must use index lookup syntax to do a search on a word or part of a word.

For more information about index lookup syntax, see "Using Indexes" in *Accessing Data*.

When to Use This Type Use Btree indexing when the field will be used for searching and when it will be used for sorting in R/LIST reports.

Use Btree indexing when you plan to use the entire contents of the field to retrieve information. Typically, it is used when indexing date, ZIP code, state, and similar single valued fields.

Use Btree indexing on fields that will be used to create sorted R/LIST reports. This will speed up report processing time.

Records in Advanced Revelation are stored in no particular order. If a user requests a sorted report, the data must be sorted before being displayed. On the other hand, data in a Btree index is maintained in sorted order. This simplifies sort requests, since there is already a list of data in sorted order. When the user requests a sorted report (specifying a Btree indexed field in the R/LIST BY criteria), the system simply reads the index and begins displaying records based on the order found in the index. There is no delay while the system sorts the data for the report.

Creating a Btree Index

To define Btree indexing in the Dictionary window, specify the file and field names and press [Shift-F1]. The entry at the *Btree (Y/N)* prompt determines whether a field will have Btree indexing. Type Y for yes. Press [F9] (Save) to add the field indexing information to the field definition. When the Dictionary window reappears, press [F9] again to save the dictionary definition and update the indexing system.

To define Btree indexing from Paint or from the Main menu, choose **Btree** from the Indexing menu. The online help, [F1] (Context) and [Ctrl-F2] (Concept), describes how to use the Btree window. The entry at the *Indexing* prompt determines whether a field is to be indexed. Type Y for yes. Press [F9] (Save) to update the indexing system.

Cross Reference Indexing

Btree and Cross Reference indexing use the same means to return keys. The distinction between the two is that Btree indexing uses the contents of the entire field as an index key. Cross Reference indexing uses the individual words (or character strings) in the field as the values in the index. You can designate any characters as delimiters.

When to Use This Type

Use Cross Reference indexing when you want to retrieve information using one word in a field as the key, rather than the entire contents of a field. Although you can use index lookup syntax to search a Btree index for a word within a field entry, it is faster to use a Cross Reference index for this type of search because the index is maintained as a sorted list of individual words. The system can go directly to the word in the sorted list. Searching for a word in a Btree index takes longer because every entry in the index must be read to determine if it contains the desired word.

Cross Reference indexing is preferred under the following conditions:

- For any field where you want to retrieve information using a portion of the field, such as one word (a name or text field)
- For any text field where text searching may be required (because of the ability to define the Stop List)

In Cross Reference indexing you can specify the delimiters (such as value mark, space, and comma) that define the elements to index separately. Each “word” in the field is recognized as any character string that is between two delimiters. Delimiters are specified when defining the index. Typical delimiters are spaces, commas, hyphens, and apostrophes.

Some words may be too common or not meaningful for indexing. You can use a Stop List in conjunction with a Cross Reference index to exclude these words. The Stop List specifies words that will not be indexed.

Whenever a Cross Reference index is created, Advanced Revelation creates a symbolic field pointing to the field being indexed. The symbolic field is named according to the format *fieldname.XREF*. For example, if you create a Cross Reference index on a field called COMPANY, the indexing system will create a symbolic field called COMPANY.XREF.

The formula for this field calls a function (XREF) that parses the data in the original field according to the Cross Reference delimiters. The XREF function also passes information about the stop list used for that Cross Reference index.

The most important fact about the XREF version of a field is that this symbolic field is in turn Btree indexed. A Cross Reference index is thus a Btree index based on a symbolic (parsed) version of the original field. The symbolic field causes each individual word in the field to be indexed individually in the Btree index.

Creating a Cross Reference Index

To define Cross Reference indexing from Paint or from the Main menu, choose **Cross Ref** from the Indexing menu. The online help, [F1] (Context) and [Ctrl-F2] (Concept), describes how to use the Cross Reference window.

The entry at the *Indexing on* prompt determines whether a field is to be indexed. Type 'Y' for yes.

To define Cross Reference indexing in the Dictionary window, specify the file and field names and press [Shift-F1]. The entry at the *Cross Reference (Y/N)* prompt determines whether a field will have Cross Reference indexing. Type 'Y' for yes. Define delimiters and specify the stop list. Press [F9] (Save) to add the field indexing information to the field definition. When the Dictionary window reappears, press [F9] again to save the dictionary definition and update the indexing system.

- When the Cross Reference index is saved, one of the Message windows will tell you:

<Updating index fieldname.XREF in filename>

The *fieldname.XREF* field is added to the file's data dictionary. It holds the delimiters and the specified Stop List that the indexing system uses when Cross Reference indexing the field *fieldname*.

Specifying Delimiters

At the *Delimiters* prompt, specify the characters used to separate words. A word ends each time a delimiter is encountered. You may specify one or more delimiters. The default delimiter is a space.

Specify delimiters based on the kind of field you are Cross Reference indexing. Depending on the field type, the following delimiters should be used:

- VM (value mark, ASCII 253)
Always use VM when Cross Referencing a multivalued field. The value mark delimiter insures that Advanced Revelation distinguishes between the end of one value and the beginning of the next, so it can index each value.
- SVM (subvalue mark, ASCII 252)
Always use SVM when cross referencing a multivalued field that may contain subvalues. This would be specified in conjunction with VM. The subvalue mark delimiter insures that Advanced Revelation distinguishes between the end of one subvalue and the beginning of the next, so it can index each subvalue within a value.
- TM (text character, ASCII 251)
Always use TM when cross referencing a text field that is multiline (not multivalued). The text mark delimiter insures that Advanced Revelation recognizes the new line mark (usually inserted when the user presses [Enter]), so it can index each new line.

A space is commonly used as a delimiter. In a multivalued field, the space delimiter is often used in conjunction with the VM delimiter. This allows you to index the individual words within each value. For example, in a multivalue name field, using both space and VM as delimiters creates indexes for the first and last name in each value.

**Specifying the
Stop List**

A Stop List specifies which Stop List the indexing system uses. The default Stop List is defined and maintained in the Index Management Environment window, which is reached by using the **Management-Environment-Indexes** options on the Advanced Revelation Main menu. Indexes are not maintained for any of the words on the Stop List. It is best to use the default Stop List, unless you are indexing a special purpose text field. Press [Shift-F2] to display the default list.

Alternatively, you can list additional stop words used in conjunction with the default list or create a custom Stop List specifically for this field.

At the *Additional stop list* prompt, enter additional stop words. This is a multivalued field; each word of the additional stop list goes on one line.

Press [F9] (Save) to update the indexing system.

Relational Indexing

Relational indexing is used when you want the system to maintain a relationship between records. It lets you use individual fields to form relationships between records in different files.

Unlike Btree and Cross Reference indexes, Relational indexes are not used by R/LIST. Their purpose is to relate detail records to a master record. This relationship is used in related windows, when activating lists of keys for detail records, and by any processes that require the related keys.

Advanced Revelation allows you to create Relational indexes on data fields, symbolic fields or parts of multipart keys. You can also specify the sort order of the indexed keys. This enables you to create Relational indexes on any data in a file, and to maintain the index in any order you choose.

**When to Use
This Type**

Use Relational indexing:

- To create a relationship between a master file and a detail file
- To set up a file structure where one record can be related to more than one other record within the same file

An invoicing application might make use of a master/detail file relationship. The CUSTOMER file is the master. Each customer master record holds information about a customer who orders goods. The INVOICE file holds detail records about individual orders. You can have many orders (invoice records) for a customer.

By creating a Relational index, a field in the customer record in the CUSTOMER file can be used to maintain a record key list (index) of all of the invoices in the INVOICE file that were generated by that particular customer. Within each data record in the CUSTOMER file is a field maintained by the indexing system. The field holds an index of all of the order records in the INVOICE file that relate to that particular customer. This Relational index is defined in the INVOICE file and is updated when INVOICE records are updated.

For more information on establishing relationships between files, see the “Planning an Application” chapter.

Creating a Relational Index

To define Relational indexing from the Dictionary window specify the file and field names and press [Shift-F1]. The entry at the *Relational (Y/N)* prompt determines whether a field will have Relational indexing. Type Y for yes. At the *Join File* prompt, enter the name of the file that will hold the index keys. At the *Join Field* prompt enter the name of the field in the join file that will hold the index keys. Press [F9] (Save) to add the field indexing information to the field definition. When the Dictionary window reappears, press [F9] again to save the dictionary definition and update the indexing system.

To define Relational indexing from Paint, or from the Main menu, choose **Relational** from the Indexing menu. The on-line help, [F1] (Context) and [Ctrl-F2] (Concept), describes how to use the Relational window.

- ☛ The name you specify at the *Join Field* (in the Dictionary Indexing window) or the *Index Field* (in the Relational Index window) must be a pre-defined F-type field in the related file. This field will become the exclusive province of the indexing system. You will not be able to edit it. The system verifies that the field you specify exists.

Sort Mode

The *Sort Mode* prompt allows you to specify the order in which the indexed data will be stored. Press [F2] at the *Sort Mode* prompt to display a list of sort mode options.

For example, you may wish to specify that the data in the index be stored in chronological order. In that case, you would choose the option **BOT** (bottom) for the sort mode. The **BOT** option puts the most recently indexed data at the bottom (end) of the index.

Mode Options for Relational Indexing

Code	Meaning
TOP	Put the most recently indexed data at the top (beginning) of the index. In other words, store the indexed data in reverse chronological order.
BOT	Put the most recently indexed data at the bottom (end) of the index. In other words, store the indexed information in chronological order.
AL	(ascending, left justified) Put the indexed data into alphabetical order.
AR	(ascending, right justified) Put the indexed data into numerical order, lowest to highest.
DL	(descending, left justified) Put the indexed data into reverse alphabetical order.
DR	(descending, right justified) Put the indexed data into reverse numerical order, highest to lowest.

- ☛ The default sort mode is BOT. If you are not concerned about maintaining a sort order for the index, accept the default option of BOT as the sort mode.

Indexing Global Files

Advanced Revelation allows you to create indexes for files that are global. Global files are those that are not associated with a particular account, and include most system files such as VERBS, COMMANDS, etc.

To index a global file, create an index as you would for any other file.

If you wish to make a file global, you can use the RENAMEFILE command from TCL. For example, to make the CUSTOMER file global, use this command:

```
RENAMEFILE CUSTOMER GLOBAL CUSTOMER
```

You must remove any indexes created for a file before making it global. After you have made the file global, you can install or reinstall any indexes you wish.

- ☛ You must attach the file (using the ATTACH command or window) after you have made it global, or you will not be able to index it.

Where the Indexes are Stored

When you create the first Btree or Cross Reference index for a file, an indexing file is created with the name *!filename*. The Btree and Cross Reference index key lists for a file are maintained in the *!filename* file. For example, the indexes for fields in the CUSTOMER file are held in the *!CUSTOMER* file.

WARNING ! Do not attempt to edit the *!filename* file. All actions taken with regard to a *!filename* file must be made through the maintenance programs that are part of the indexing system. Any attempt to make changes outside the indexing system can corrupt the file.

Unlike Btree and Cross Reference indexes, Relational indexes are not stored in the *!filename* file. Each Relational index is held in a field in the master file. Once this field is defined to hold the index record keys, it is protected. It is only updated by the indexing system.

Using Quickdex

In addition to the three kinds of indexing that can be defined within a file, a special file indexing feature, called Quickdex, is available in the Dictionary window. Quickdex creates and maintains a sorted list of the record keys for a data file or a dictionary file. These sorted lists save time when using R/LIST and popups in table mode.

What is Quickdex?

Quickdex maintains a sorted list of keys to a file. Unlike Btree or Cross Reference indexes, which are maintained in indexing files, Quickdex is a special record in the file being indexed. Once Quickdex is specified in a file, the next time a read or write to the file occurs, the Quickdex key list is created. Thereafter, the indexing system will update and maintain the sorted key list.

The Quickdex facility in Advanced Revelation allows you to specify how record keys are to be sorted. Record keys can be maintained in a left-justified (alphabetic) sort or in a right-justified (numeric) sort.

Quickdex bases its sort order on the justification you have specified for the key field in a file. For example, if the key field is left, center, or text justified, the Quickdex will be left justified. (The key field is a field with a field number of zero.) If the key field in a file is right justified, then the Quickdex for that file will also be right justified. When the Quickdex facility creates a record key list that is right justified, it is identified as a Rightdex.

The sorted key list is stored in a hidden record called %RECORDS%. The %RECORDS% record is written to the data file or the dictionary file. Do not delete the %RECORDS% record from your file unless the sorted list has become corrupted. Any time the %RECORDS% record is deleted, it will be recreated when the next read or write to the file occurs.

- ☛ %RECORDS% has a maximum size of 64K. Therefore, you should not use Quickdex on a data file that has very long keys or numerous records. The 64K limit translates into roughly 8,000 seven character keys (that is, 8,000 data records, each containing a seven character key).

Reasons to Use Quickdex on a Data File

Use Quickdex on a data file to make R/LIST more efficient when a SORT or SELECT of an entire file is requested (and no field criteria are specified). If Quickdex is specified for a data file, a sorted list of all record keys is available. R/LIST can use the Quickdex immediately to locate and list the records. There is no need to read through the entire file, build a list of record keys, and then sort them.

Creating a Data File Quickdex

In order to establish a Quickdex, display the Dictionary window using the **Tools-Files-Dictionary** menu options. A softkey [Shift-F3] in this window sets up the Quickdex.

Enter the desired file name and press [Shift-F3] to specify a Quickdex index of the data portion of a file.

If a key field is already displayed in the Dictionary window when you press the Quickdex softkey, the Quickdex will be created according to the justification of that field. If the file has more than one key field, be sure to display a key field with the justification you desire for the Quickdex before you press [Shift-F3].

If no field is displayed at all in the Dictionary window when you press [Shift-F3], the Quickdex process will prompt you to specify whether you want the keys sorted left or right justified. The record keys will be sorted according to your response.

A message will display confirming that a Quickdex or Rightdex modifying filing system has been added to the file.

- ☛ After you have installed Quickdex on a file, you will be prompted by the system to create the Quickdex index. Answer Y at this prompt in order to build the Quickdex. The Quickdex should be built before you add records to or delete records from the file.

Reasons to Use Quickdex on a Dictionary File

Use Quickdex on a specified dictionary file to make the display of system popups that use record keys more efficient. @ID and @IDS popups (any system popup key display) will be faster.

Creating a Dictionary Quickdex

The dictionary Quickdex indexing option is only available through a softkey in the Dictionary window. After entering the desired file name, press [Shift-F4] to specify a Quickdex index of the dictionary portion of a file.

The Quickdex process will prompt you to specify whether you want to install a Quickdex or Rightdex modifying filing system. The record keys will be sorted according to your response.

A message will display confirming that a Quickdex or Rightdex modifying filing system has been added to the file.

Removing Quickdex from a Data File

[Shift-F3] acts as a toggle to switch data file Quickdex on and off. After entering the desired file name, press [Shift-F3] to remove a Quickdex index of the data portion of a file.

Messages display to confirm that a Quickdex or Rightdex modifying filing system will be removed from the file.

Removing Quickdex from the Dictionary

[Shift-F4] acts as a toggle to switch the dictionary file Quickdex on and off. After entering the desired file name, press [Shift-F4] to remove a Quickdex index of the dictionary portion of a file.

Messages display to confirm that a Quickdex or Rightdex modifying filing system will be removed from the file.

Determining if Quickdex is Specified for a File

To determine which files have a Quickdex index, select the **List Volume** option on the File Tools menu, or enter LISTVOLUME at The Command Level (TCL). Where a data or dictionary file has a Quickdex, "QUICKDEX.MFS" or "RIGHTDEX.MFS" will appear in the column labeled "Modifying Filing Systems".

Indexing System Maintenance

This section explains how Advanced Revelation maintains the indexes you have defined. It also covers some critical information about moving data files and indexes from one volume to another.

Background Processing by the Indexing System

If you change a record using any Advanced Revelation process, the indexes will be updated automatically. The update of the indexes occurs as a background process. This makes it possible to write index transaction records that update the indexes when you are not busy.

When a record is written or read, Advanced Revelation's filing system checks the dictionary to see whether indexing is defined. For each field index defined in the record, the filing system writes a transaction record to the !INDEXING file. Each transaction record specifies the index to update, the new field value and the record key (location) for that value. The indexing background process uses the !INDEXING file to store the index transaction records until the time is available to update the indexes.

- It is absolutely essential that all network users share the same !INDEXING file. Otherwise, the timing sequences and integrity of the index files will be corrupted.

Locating Indexing Specifications

Advanced Revelation's indexing system finds the indexing specifications for a field in the dictionary record for that field. If the field is Cross Reference indexed, there are two dictionary records used to define that index. The specifications for delimiters and the stop list are maintained in a dictionary record named *fieldname.XREF*.

System Index Updating

The Advanced Revelation indexing system uses any workstation that is not currently receiving keyboard input or running a process to process the transaction records in the !INDEXING file.

The information in a Btree or Cross Reference transaction record is written to the !*filename* file (the actual data file index). When updating a Relational index, the information in the transaction record is written to a field in a related record. When the indexing system updates a Quickdex key list, it updates the %RECORDS% record in the appropriate data file or dictionary.

Table F7-1
Index Locations

Index Type	Location of Index
Btree	!filename
Cross Reference	!filename
Relational	field in a related record
Data Quickdex	%RECORDS% in data file
Dict Quickdex	%RECORDS% in dict file

Managing Background Processing

Use the Index Management Environment window to adjust the indexing functions for each user environment. On a network, you can set up environments so that one station takes over the primary function of indexing. This is done by setting the *Delay before indexing* prompt to 1. If the *Delay before indexing* prompt is set to 0 then that station will not update indexes.

You can also set the environment so that it will automatically force the completion of transactions when indexed fields are referenced in an R/LIST sentence or when a filter is used. The !INDEXING file transactions are completed and pending transactions for that file are processed prior to processing the SELECT or LIST sentence.

To access the Index Management Environment window from the menu, select **Management-Environment-Indexes**. The online help, [F1] (Context) and [Ctrl-F2] (Concept), describes how to use the window.

Deleting an Indexed File

When you delete a file with indexing, the system will automatically flush all outstanding index transactions for that file before deleting it and the respective !filename file.

Special Considerations When Working with Indexed Files

If a file is indexed, Advanced Revelation will update the index every time a changed record is written to the file. No matter which process you use to update the file — a window, the Editor, or your own R/BASIC program — Advanced Revelation will automatically maintain the index. Operations that affect files and not records do not recognize the indexing system.

If you wish to use REMAKEFILE, or RENAME file you must remove indexing for the file before using the command. The FILECOPY command will remove the indexing as part of its operation. *For this reason, FILECOPY is not a good choice for backing up indexed files.*

If you do use REMAKEFILE or RENAME on a file, follow this procedure:

1. Use Index Update Option 6, **Remove Indexing from a File** to delete the index.
2. Use the FILECOPY or RENAME command.
3. Redefine the indexes for that file, then update the new indexes.

Changing the Location of Indexed Files

Advanced Revelation has a standard set of volumes it uses for operation. One of these, INDEXING, ties all the individual indexes together using a master indexing file. This file then tracks all of the indexing functions for the entire system.

Indexing is a system-wide resource and a specific index is associated with a specific location. To guarantee that data is secure and that indexing functions properly, no two volume directories should have the same name. The volume directory takes its name from the time and date when the first file is created on the volume or from the name specified in a NAMEVOLUME command. You *must not* use duplicate names for your volumes. The combination of filename and volume name must be unique in your installation. It is recommended that you leave the time-date stamp on the volume directory or come up with a system for unique volume directory names.

- ✎ When a subdirectory is copied to another location, there will be two places in your installation that have the same filename and volume directory name. Always keep your indexed files on your hard drive while they are used.

If you want to transfer data from one location to another outside of Advanced Revelation:

1. Remove your index.
2. Use the operating system copy utility to copy the files to the new location.
3. If you moved the REVMEDIA file, use NAMEVOLUME to give the file a unique name in the new location.
4. Recreate the indexes in the new location.

Reviewing Index Specifications

This section describes the system index reports and how to display them. These reports show you what indexes are defined.

Using List Index

Use List Index to print index reports to the screen. Enter the LISTINDEX command at the TCL level, or select the option on the File Indexing menu.

Advanced Revelation searches for all files that contain an index and displays a popup listing them. The popup shows the types of indexing in each file, and any related files.

Viewing Indexed Fields in a Selected File

You can get more detailed field indexing information about any file by selecting the file from the initial Listindex popup. A message window will display:

```
<Indexed file:  filename
Do you wish to view indexed fields?>
```

Enter Y to view a popup that lists the indexed fields in a specific file. The popup shows the types of indexing in the file. In the “Type” column:

- *Related to* means that the field writes keys to the file and field shown in the “Detail” column
- *Related from* means that the field holds the indexed keys for the file and field shown in the “Detail” column
- *CrossRefing* means that the field contains the delimiter and Stop List specifications for the field listed in the “Detail” column
- *CrossRefed by* means that the field is indexed using the specifications in the field listed in the “Detail” column

☛ This popup is also available in the Dictionary window. Press the softkey [Shift-F2] to see a listing of the indexed fields for the specified file.

**Viewing Indexing Detail
for a Selected Field**

When using List Index, you can see which kinds of indexing have been specified for a field by selecting the field from the Indexed Fields popup. A message window will display:

```
<File:  filename   Field: fieldname
Do you wish to view Indexing detail?>
```

Enter Y to view a popup that lists the kinds of indexing specified.

- ✎ This popup is also available in the Btree, Cross Reference, and Relational Index windows. Press the softkey [Shift-F1] to review the indexing specifications for a specified field. It is a good idea to look at this display prior to defining new indexing characteristics for the field.

Printing Index Reports

You can specify the following syntax at The Command Level (TCL) to route indexing reports to the printer:

Command	Meaning
LISTINDEX (P)	Print a list of all files with indexing
LISTINDEX <i>filename</i> (P)	Print fields with indexing for a file
LISTINDEX <i>filename fieldname</i> (P)	Print indexing detail for a field in a file

Updating and Rebuilding Indexes

This section describes the correct ways to update, recreate, and delete indexes. It explains all of the Update options available on the File Indexing menu. These functions are used primarily by the system administrator. To maintain the integrity of your indexes and ensure that the system can properly index files, any change to the indexing files should be made through the Update options.

Index Update Choices

To force index updates to a data file, choose the **Update** option on the File Indexing menu. A popup of update choices will display. These choices let you update, rebuild, or remove an index.

The distinction between updating and rebuilding an index is this:

- **Update** checks the !INDEXING file to see if there are any pending index transaction records for a specified file. If there are, it indexes them immediately. It does not erase the old index and build a new one.
- **Rebuild** erases the current index for a specified field (or fields) from the !*filename* index file. Then it reads all the records in the file and recreates the index.

Updating Indexes

The update process updates the indexes using the !INDEXING transaction records for a specified file or field. This guarantees that the index is up to date.

The three index update options are described below.

Update All Indexes When you choose Option 1, this message window displays:

```
<Updating Index "ALL" in "ALL">
```

Your workstation takes on responsibility for processing all of the pending transactions in the !INDEXING file for all files that are currently attached. Your work station is dedicated to updating the indexes of attached files with these transactions.

Update Indexes In a Specific File

Use this option to update field indexing for a specific file. Enter the file name. At the *Field* prompt accept the default "ALL" to update every field index in the file. You can have more than one kind of indexing on a field. If you need to update only one type, specify that type at the *Index Type* prompt. To update all of the field's indexes, type in "ALL". The default index type is Btree.

When you press [F9] (save), this message window displays:

```
<Updating Index "ALL" in "filename">
```


**Update a Specific
Field Index**

Use this option to update a specific field index in a file. You might use it when you are running a report and know that many transactions have been entered and are pending in the !INDEXING file. To insure that all of the latest records are available for the report, flush the transactions indexed on this field and make sure that they were added to the index.

If you define a new field index in an existing data file and do not update the initial index at that time, the update processing can be initiated with this option. (Alternatively, the background indexing system will update the index for you as time becomes available.)

Enter the file and field name. Specify the index type or type "ALL" at the *Index Type* prompt. The default index type is Btree.

Rebuilding Indexes

You only need to use the Rebuild process under one of the conditions listed below:

- A Group Format Error causes loss of data in an indexed data file.
- The indexed data file is lost or destroyed by an operating system process.
- You suspect that an index is inaccurate or corrupted and does not contain all record keys.
- On a network, you discover that there are two files with the same name on two volume directories with the same name.

The Rebuild process clears a specified type of index for a field or it clears that type of index for all the indexed fields in a file. The key list is erased from the !*filename* index file. If you are rebuilding all the indexes for a file and Quickdex is specified, the %RECORDS% record is erased from the file as well. Then Rebuild reads all the records in the data file and recreates the indexes that were erased.

It may take considerable time to rebuild indexes in a large data file. You may want to run the rebuilding process at night or during a quiet time on the system.

The two rebuild options are described below.

**Rebuild Index
for a Field**

Use this option to erase and recreate the index for a specified field. Enter the file and field name. Specify the index type or type "ALL" at the *Index Type* prompt. The default index type is Btree.

**Rebuild All Indexes in a
Specific File**

Use this option to erase all of the indexes for a file and then rebuild them. The rebuild program reviews every data record for that file and indexes each one. This option prompts for a file name, and defaults to "ALL" for field names and index types. You can specify a field and/or an index type if preferred.

Restrictions to Index Update Processing

Only one user at a time can perform index updating. If more than one user tries to perform index updating at the same time, only the first user will be successful. Users who attempt to update an index while another user is doing so will be “locked out” of the index update process. Advanced Revelation will display a message warning them that there will be a delay before their index update request can be processed. If they choose, they can cancel their request.

- ✎ The restriction described above only concerns index update processes. New indexing transactions can be created at any time.

Indexing Processes Restricted to One User

Here is a list of the index updating processes that can be run by only one user at a time:

- establishing a new index and specifying that it should be updated immediately
- using any of the Update or Rebuild options available through the **Update** option on the File Indexing menu
- using the INDEX command at TCL to update an index
- executing the INDEX.FLUSH subroutine in an R/BASIC program

In addition, while Advanced Revelation’s background index processing is performing an index update, no user can run an index update process.

The Update Delay Message

If one of the processes listed above is already running when a user tries to perform an index update, the following message displays:

```
Another workstation is currently updating
indexes.
```

```
The index update you have requested will
proceed when the other workstation's update
is complete.
```

```
You may skip the update you have requested
by pressing [Esc]. Press any key to
continue.
```

When you see this message, you can wait for the current updating process to end. When it does, the message will disappear and your updating process will begin. Alternatively, you can press [Esc] to cancel the update process you wanted to run and try again later.

**Avoiding Index
Processing Delays**

Consider the impact on other users before directing Advanced Revelation to update a new index or rebuild an existing index for a large data file. When working with a large file, updating may take a long time. During that time, other users will not be able to perform any process that updates an index. If they try to do so, the update delay message will display and interrupt their processing. For large files, it is best to run a batch index updating process at night or during a quiet time on the system.

**Updating and
Background Indexing**

By its very nature, background indexing is an intermittent process. If the background indexing process determines that another index update process is already underway, it will not try to perform an update at that time. And, since there is no delay, it will not display the update delay message that is shown above.

Deleting Indexes

Index specifications and the indexes themselves are deleted through options on the File Indexing menu. They can be deleted through the Dictionary window, by using the [Shift-F1] softkey to access the Dictionary Index window.

In the Dictionary window, type in the name of the file and the field that is indexed. Press [Shift-F1] to display the Dict Index window. Then move the cursor to the prompt (or prompts) that names that index and change the status from Yes to No. Press [F9] to save the change.

At the File Indexing menu, choose the menu option for the type of index you want to delete. When the indexing window appears, type in the name of the file and the field that is indexed. Then, enter N at the *Indexing on* prompt to change the status of the indexed field from Yes to No. Press [F9] to save the change.

When you change the indexing status from Yes to No, the following message will display:

```
<The index for this field will be deleted. Are you sure Y/N?>
```

To delete field index specifications and all indexes for this field, enter Y. If the field is Cross Reference or Btree indexed, this removes the indexed values and record keys from the *!filename* indexing file. If the field is Relational indexed, this removes the index keys from the related field in each record in the related file.

Remove Indexing from a File through Update

There is another way to delete indexes and index specifications. Choose **Update** from the File Indexing menu. When the Update options are listed, choose Option 6, **Remove Indexing From a File**.

This option is used to delete all indexing in a file. When used, it performs all of these tasks:

- Erases any indexes in the *!filename* indexing file.
- Erases the %RECORDS% record from the specified file.
- Removes all field indexing instructions from the field records in the dictionary.
- Turns off Quickdex for the specified file

Developer Notes for Indexing

Advanced Revelation makes information about the indexing process available to developers.

Index Transaction Code

Advanced Revelation generates a custom index program for each indexed file. This method makes index transactions both efficient as well as accessible to developers.

Anytime you create an index, Advanced Revelation generates and compiles an R/BASIC program that is used to create index transactions for that index. This is known as the index transaction code.

Index transaction code is called whenever a record update is made that involves an index update. For example, suppose the COMPANY field of the CUSTOMER file is indexed. If you write a new record to the CUSTOMER file, an index transaction is created that notes that the company index must be updated with information from the new record.

Index transactions are temporarily stored in the !INDEXING file. Later, when your computer (or another computer on your network) is idle, the index transaction is processed, and the actual index update made.

The index transaction code contains all the logic required to maintain the index transactions for any one file. For example, if you index the fields COMPANY and ZIP in the CUSTOMER file, Advanced Revelation will create index transaction code containing logic that specifically maintains the index transactions for these two fields. If you add or delete indexes, the index transaction code is regenerated and recompiled.

Locating Index Transaction Code

When Advanced Revelation creates index transaction code, it compiles it, and then writes it to the appropriate index (!*filename*) file. The object code is written to a record with a record key of "!" (exclamation point).

For example, if you index a field in the CUSTOMER file, the index transaction object code is written to the record "!" (exclamation point) in the !CUSTOMER file. If you change, add, or delete indexes for the CUSTOMER file, this object code is regenerated and written again.

- ☛ If you delete the index transaction code, Advanced Revelation will need to regenerate the code in order to maintain the index, increasing the index processing time.

Index Transaction Source Code

If you create a record in the appropriate index (!*filename*) file with a key of "!!" (two exclamation points), Advanced Revelation will capture the index transaction source code when it is next generated.

You can study this source code to learn more about the way Advanced Revelation maintains index transactions. You can also make changes to this code, and then use your version of the code. When index updates are next required, your version of the index transaction code will be called.

For example, using the editor, create an empty record "!!" (two exclamation points) in the !CUSTOMER file (the index file for CUSTOMER). When you index a field in the CUSTOMER file, the index transaction object code will be written to the !CUSTOMER file under the name "!" (exclamation point), and the corresponding source code will be captured and written under the name "!!" (two exclamation points).

In order to modify and update the "!" record, several steps are required. First, the source code in "!!" must be modified. You must then produce the object code of that item in the format used for symbolic dictionary items. This is required because the "!" record is called like a dictionary record.

One way to make this update is to modify the source code in the “!!” record, and then to run a program such as this one:

```
* creates new ! record in !file
FILE = ""
CALL MSG("Which file?","R",FILE,"")
OPEN "!!":FILE TO INDEX.FILE ELSE
CALL MSG("!!":FILE:" can't be opened!","", "", "")
  STOP
END
READ SOURCE FROM INDEX.FILE, "!!" THEN
  NEW.UPDATE.PGM = "S"
  CONVERT @FM TO @VM IN SOURCE
  NEW.UPDATE.PGM<8> = SOURCE
  WRITE NEW.UPDATE.PGM ON INDEX.FILE, "$!!"
  PERFORM "DICTCOMPILE !!":FILE:" $!!"
  READ OBJECT FROM INDEX.FILE, "$!!" THEN
    WRITE OBJECT ON INDEX.FILE, "!"
  END
END ELSE
  CALL MSG("No !! source code available!","", "", "")
END
```

Warning! Modify the index transaction code at your own risk. Because you can affect the way indexes are updated, errors you introduce into the index transaction code can corrupt your indexes, making them unusable. Before implementing any changes to the code, you may wish to back up your indexes, as well as the original versions of the index transaction source and object code. You should also be aware that each time you modify (add or delete) an index, the code in the “!” and “!!” records will be overwritten.

Chapter F 8

Building Formulas

This chapter explains how to use popups in the Formula window to compose an R/BASIC formula. It describes the options available through these popups and how special “trigger” words are used to reference them. This chapter also contains a glossary of trigger words giving the meaning and use of each one.

What is an R/BASIC Formula?	F 8 . 1
Accessing the Formula Window	F 8 . 1
Glossary of R/BASIC Formula Trigger Words	F 8 . 4



What is an R/BASIC Formula?

R/BASIC formulas perform calculations and make the results of those calculations available to Advanced Revelation processes. The value that a formula yields is stored in a system variable called @ANS (answer). @ANS always contains the value generated by the most recently executed formula.

Using Formulas

In Advanced Revelation, symbolic fields (symbolic dictionary records) are defined by use of a formula. A formula consists of one or more lines of R/BASIC code that, when executed, return a value for the symbolic field.

The R/BASIC code that defines a formula can consist of almost any logic. Typical examples include formulas that calculate a sum or product of the contents of other fields in the record (for example, an invoice total); formulas that string together (concatenate) values in fields, perhaps together with literal values (for example, a mailing label format); and formulas that extract (translate) information out of related files (for example, a product name based on a known product code).

Whatever logic is used to calculate the value for a formula, the final value must be loaded into a system variable called @ANS. For example, a simple formula might consist of the logic @ANS = {SUBTOTAL} + {TAX}, in which the current values of the SUBTOTAL and TAX fields are added and the resultant total assigned to the system variable @ANS.

Formulas can contain up to 34K of source code as their definition. Within that limit, they can take up as many lines as required to define the logic of the definition. The R/BASIC code for a formula is stored as a field in the appropriate dictionary record. For more information about dictionaries, see the “Advanced Revelation Dictionaries” chapter in *Filing Systems*.

Accessing the Formula Window

You may reach the Formula window from the TCL, Dictionary, Definefile, and Batch Update windows or from Paint by way of the Dictionary.

To enter from the TCL window, either press [Shift F3] or press [F6] (Softkeys) and select Option 3 from the TCL softkey popup. This will access the Expression window, which will invoke the Formula window from the *Field name* prompt.

To enter from the Dictionary window, press [F2] (Options) at the *Formula* prompt.

To enter from the Definefile window, press [Shift-F6] or press [F6] (Softkeys) and select Option 6 from the softkey popup.

To enter from the Batch Update window, press [F2] (Options) at the *Update Formula* prompt.

To enter the Dictionary window from Paint, press [Shift F10] or press [F6] (Softkeys) and select Option 10 from the softkey popup. From the Dictionary window, press [F2] (Options) at the *Formula* prompt.

Accessing the Formula Assistance Popups

The Formula window has built-in popups that may be displayed to help define the formula being built. These assistance popups may be used to:

- prompt you through the creation of all parts of an R/BASIC formula
- provide help for a specific part of a formula
- append additional expressions to an existing formula

For comprehensive help For popup help with all parts of formula composition, press [F2] (Options) from within the Formula window. After selecting the kind of formula to create, lower case trigger words associated with it are written to the window. Each trigger word represents an element in an R/BASIC formula. Trigger words identify the components of a formula and the sequence in which they must appear. An underlying program reads the lower case trigger words and determines which popup to display next.

For partial help You may only need help to write a portion of a formula. In this case, enter the lower case trigger word that represents the element of the formula in question, and press [F2] (Options) to invoke the appropriate assistance popup. The assistance popups related to a formula segment may be used as often as needed.

To add expressions to an existing formula For help in adding additional expressions to an existing R/BASIC formula, press [F2] (Options) to access the **more** popup. This popup allows you to select the type of operation to be added. Trigger words are appended to the formula and are used to call the appropriate assistance popups.

☛ Once an assistance popup is called, the assistance process continues until explicitly turned off. To turn off assistance, press [Esc] while a popup is displayed. To turn it back on, press [F2] (Options) at a trigger word.

Using the Popups

The Formula assistance program reads the trigger words in the Formula window and displays the popups associated with those words. Each trigger word calls a popup or series of popups used to aid in writing a segment of an R/BASIC formula.

The popups prompt for information and present more choices related to the current segment of the formula. Popups may write more trigger words, or a value and

trigger words, back to the collector window. A trigger word is written back to the window any time more information is needed or another option could be included at this point in the formula.

By responding to the popups, the trigger words in the R/BASIC formula are replaced with real expressions. Responses are written back to the Formula window in place of the trigger word.

You have the opportunity to skip selections (that is, to not include that element in your formula). Another option allows you to undo the last change made to the R/BASIC formula, and return to a point just prior to that operation.

When a completed formula is saved, it is immediately compiled. Compilation checks the validity of the formula, and, if it is valid, stores the object code for the formula in the appropriate file dictionary. Invalid formulas are not saved.

Choosing a Formula Type

The first assistance popup (when constructing a new formula) offers five formula options. To step through all formula possibilities, choose the last formula listed.

@ANS = synonym A synonym is merely another name for an existing dictionary item; a common use of *synonym* is to create dictionary records with shorter names than the original dictionary record. For example, the synonym CO with the formula

```
@ANS = {COMPANY}
```

would return the value of the field COMPANY in @ANS whenever the dictionary item CO was referenced.

@ANS = constant Any numeric value or string of characters may be used as a formula constant. For example, the formula

```
@ANS=3.14159265
```

(an approximate value of pi) will assign the constant value of pi. The formula

```
@ANS="ABC"
```

will assign the constant text value "ABC".

@ANS = join Data from any Advanced Revelation file may be accessed with a formula (subject to security restrictions). This is sometimes known as "joining". The formula

```
@ANS=XLATE ("CUSTOMER", "100", 1, "X")
```

would extract field number 1 for customer number 100 from the CUSTOMER file.

@ANS = field formula This is used to perform a calculation on fields within a file. For example, the formula

```
@ANS = {FIRST_NAME} : {LAST_NAME}
```

might be used to calculate a complete name.

@ANS = value Any other desired complex R/BASIC formula that does not match one of the 4 above types may also be specified. An example would be

```
@ANS = LEN (@RECORD)
```

which could be used to determine the length of each data record.

Glossary of R/BASIC Formula Trigger Words

The trigger words in this glossary are organized in alphabetical order; this will not be the order in which they would be encountered while using the R/BASIC assistance popup to write an R/BASIC formula.

Some master trigger words represent a subset of other, lower-level trigger words. When such a master trigger word is selected, it writes back to the Formula window additional words used to specify particular formula terms.

Although not required, a familiarity with R/BASIC is useful in the construction of formulas. You may wish to refer to the *R/BASIC* section for details regarding some of the features described in this glossary.

at_variable

Use **at_variable** to use the current record, the current record key or the value returned by the formula. The variables @ID, @RECORD, and @ANS are available from the popup.

- @ID Key from the current record.
- @RECORD Data in the current record.
- @ANS The value returned when the formula is done.

For instance, you could check the value of a field in the current record by searching the contents of @RECORD.

For information on special variables, see “Special R/BASIC Variables” in the “Programming with R/BASIC” chapter in *R/BASIC*.

comparisons

Use comparisons to compare one value to another or to compare a value to a pattern. One value may be equal to, greater than, or less than another. The comparison word is placed between the two values to test for the specified condition. This keyword is also used to test for alphabetic, numeric, or specific patterns of information within a value. For instance, a ZIP code is five digits.

The operators are:

- EQ Are the values EQUAL?
- NE Are the values NOT EQUAL?
- LT Is value one LESS THAN value two?
- GT Is value one GREATER THAN value two?
- LE Is value one LT OR EQUAL TO value two?
- GE Is value one GT OR EQUAL TO value two?
- MATCH Does the value match a pattern?

For information on comparison operators and pattern matching, see “Comparison Expressions” and MATCH in the “R/BASIC Command Reference” chapter in *R/BASIC*.

concatenation

Use **concatenation** to append one string or dynamic array to another. Concatenation may be done for simple strings using the : operator, or for strings that contain value marks (multivalues) using the ::: operator. For information on concatenation, see “Arithmetic Operators” and “Special Multivalued Operators” in the “R/BASIC Command Reference” in *R/BASIC*.

constant

Use **constant** to enter a number or a literal string in the formula. Constants may be numeric (for example, 123.4567) or alphanumeric (for example, JOHN SMITH). For information on constants, see “Constants” in the “R/BASIC Command Reference” in *R/BASIC*.

field_comparisons

(see **comparisons**)

field_concatenation

(see **concatenation**)

field_formula

Use **field_formula** to perform an operation using two fields. The field_formula popup writes the trigger words “field operator field” to the Formula window. The field trigger word displays a list of field names. The operator trigger word specifies the type of operation used with the fields. For more information, see the field and operator trigger words.

field_logical

Use **field_logical** to combine two fields in a logical expression. A logical expression is used to determine whether a condition is met. The word OR is used for a logical expression that is true if either of the fields meets the test condition. The word AND is used for a logical expression that is true if both fields meet the test condition. For more information, see AND and OR (Logical Expressions) in the “R/BASIC Command Reference” in *R/BASIC*.

field_math

Use **field_math** to perform a calculation on two fields. These operators are available:

Operator	Operations
-	subtraction
+	addition
*	multiplication
\	division
---	subtraction of a multivalued string
+++	addition of a multivalued string
***	multiplication of a multivalued string
\\	division of a multivalued string
**	Raise to power
^	Raise to power

For information on these operators, see “Arithmetic Operators” and “Special Multivalued Operators” in the “R/BASIC Command Reference” in *R/BASIC*.

formula

Use **formula** to display the **Formula** popup. The formula types are:

- @ANS = synonym** This formula type is used to create an alternate dictionary record for a field.
 - @ANS = constant** This formula type is used to assign a constant value to a field.
 - @ANS = join** This formula type is used to display the value of a field in another file.
 - @ANS = field_formula** This formula type is used to perform an operation on two fields.
 - @ANS = value** This formula type is used to assign the value of an R/BASIC expression to a field.
- For more information, see “Choosing a Formula Type” earlier in this section.

function

A **function** is a pre-defined calculation that can be called from your formula. For example, R/BASIC includes functions to calculate sums of numbers, square roots, lengths of strings, the current date or time, etc. In most cases, you will provide in parentheses the value(s) you wish the function to calculate.

For more information on this and related subjects, and for a list of available R/BASIC functions, see “Internal Functions” in the “R/BASIC Command Reference” in *R/BASIC*.

- ☛ Wherever you see “val” the value trigger word is written to the formula window and the associated popup is displayed. The popup allows you to select the type of operation used to determine the value.

logical

(see **field_logical**)

math

(see **field_math**)

more

Use **more** to extend a formula by adding a section that performs logical, comparison, concatenation, math, or substring operations. For more information, see **field_logical**, **comparisons**, **concatenation**, **field_math**, and **substring**.

more_function

Use **more_function** to extend a formula that has a function. **more_function** writes a ; (line continuation character) and the trigger word *function* to the formula window. For more information, see ; (Multiple Statement Operator) in the “R/BASIC Command Reference” in *R/BASIC*.

operator

Use **operator** to select the type of operator for a formula. For more information on types of operators, see the **field_math**, **concatenation**, **comparisons**, and **field_logical** trigger words.

substring

Use **substring** to extract characters from a string of characters. For more information, see “String Operators” and [] (Bracket) in the “R/BASIC Command Reference” in *R/BASIC*.

synonym

Use **synonym** to define a new name for an existing dictionary record. For more information, see { } (Braces) in the “R/BASIC Command Reference” in *R/BASIC*.

value

Use **value** to define an R/BASIC expression. When the expression is resolved, the value remains in the place where the expression is defined. For more information, see “Arithmetic Expressions” and “Comparison Expressions” in the “R/BASIC Command Reference” in *R/BASIC*.

Chapter F 9

Using Codes and Commands

Codes and commands allow you to perform operations that would normally require R/BASIC programs. The code specifies the type of operation to perform and the command specifies the operation. Codes and commands can call windows and menus, perform TCL commands, call R/BASIC programs, call popups, call help, and perform other operations.

Where to Use Codes and Commands F 9 . 1

The Codes and Commands F 9 . 5

The Code and Command Options F 9 . 12



Where to Use Codes and Commands

Code and command processes are available as part of the definition of several types of template records. Codes and commands can be defined for windows, forms, menus, macros, accounts, users, and keystroke capture sequences.

When designing a window or form, codes and commands can be defined in the Prompt and Default windows. The Paint menu provides options that call windows that use codes and commands. These windows are Customize, Process, Relations, Softkeys, and Collector.

Codes and Commands in the Prompt Window

The Prompt window in Paint allows the specification of codes and commands at these prompts:

- Options** Use a code and command here to specify the response to pressing the [F2] (Options) key for this prompt. For instance, the code P followed by the command @FILE displays a popup of the attached files in response to pressing the [F2] key.
- Detail help** Use a code and command here to define the help displayed when [F1] (Context help) is pressed. A common help code is HD, which displays the contents of the *Description* prompt found in either the dictionary record associated with the current prompt or in another dictionary record in the current file.
- Pre-prompt** Use a code and command here to call a process just before the cursor enters the prompt. For example, the S code might call an ADDRESS subroutine that reads a related address and writes the address to the entry line. The address could be edited when the cursor is placed on the entry line.
- Post-prompt** Use a code and command after the cursor leaves the entry line to perform an operation on the typed data. For example, the S code might call an INV.UPDATE subroutine that updates the inventory when a part is sold.
- Invalid** Use a code and command to perform a special operation when the typed data is not correct. For example, using the HM code with a command of "Enter the salesman information using the Personnel window" calls a message window that displays the message and requests a response when the salesman's initials are not verified.

The Default window in Paint is used to define the default information that is copied to new Prompt windows. This information can include the default codes and commands.

Codes and Commands in the Customize Window

The Customize window in Paint allows the specification of codes and commands at these prompts:

- | | |
|--------------------------|---|
| Concept | Use a code and command here to call a help window that is displayed when you press [Ctrl-F2] (Concept). For example, the code HF and the command GL_HELP, TRANSACTION displays the TRANSACTION help record in the GL_HELP file. |
| Detail help code | Use a code and command here to display a default help record. This allows you to define [F1] (Context) help for those prompts that have no specific help message of their own. |
| Query/Index popup | Use a code and command here to specify what is to happen when Query returns a select list. This permits you to modify or refine (but not clear) the select list. For example, your code and command might call a multiselect popup from which the user can edit the list returned by the Query process. |
| View R/LIST | <p>This code and command allows you to substitute your own window or process for the default RLIST window that is provided. If you want to replace the RLIST window with your own window or process, enter a code and command to specify the action that should occur when the user presses [Alt-V].</p> <p>☛ You can only call a substitute process or window if you did not use the <i>View R/LIST template</i> prompt to specify a default R/LIST sentence.</p> <p>If you create a custom process or window that will generate an R/LIST sentence, put the finished sentence into the variable @PSEUDO. When your process terminates, View will execute whatever it finds in @PSEUDO as an R/LIST sentence. If @PSEUDO remains null, View will simply return to the original application window.</p> |

Codes and Commands in the Process Window

The Process window in Paint allows the advanced designer to add special processes to normal window operation through the specification of codes and commands at these prompts:

- Pre-process** Use a code and command to perform a special process before entering the window, dividing the window template, reading a record, saving a record, deleting a record, or executing a query.
- Post-process** Use a code and command to perform a special process after leaving the window, dividing the window template, reading a record, saving a record, deleting a record, or executing a query.
- Replace-process** Use a code and command to replace the read, save, or deletion of a record with a process of your own.
- Perpetual-process** Use a code and command to perform an operation between prompts.

Codes and Commands in the Relations Window

The Relations window in Paint allows you to associate other windows with the window being defined. Operations can be performed before entering the related window and after leaving the related window through the specification of codes and commands at these prompts:

- Pre-window** Use Pre-window to perform a process before entering the related window.
- Post-window** Use Post-window to perform a process after leaving the related window. For example, you might use the code E and the command `RUN BP PRINT_INVOICE` to run an R/BASIC program that prints an invoice when leaving the window.

Codes and Commands in the Softkeys Window

The Softkeys window in Paint allows you to define an operation for each of twenty function keys that are associated with the window. The operation performed by each key is defined by a code and command. For example, a CUSTOMER window might use the code E and the command `RLIST QUERIES PHONE_LIST` to run a predefined report that displays a phone list on the screen.

Codes and Commands in the Collector Window

When designing a collector window in Paint, the Collector window is used to define the operation of the window. A collector window is used by advanced developers for gathering user input for a program. The information gathered by a collector window can be passed to related programs through the specification of codes and commands at these prompts:

- Setup** Use a code and command here to perform an operation when the window is displayed. For example, the code S and the command LOAD_CURRENT_DEFAULTS might be used to run a subroutine that writes default information to the prompts before the window is displayed.
- Save** Use a code and command here to specify an operation when the user presses [F9] to save the current record.

Codes and Commands in the Environment Management Windows

The following windows, accessed from the Environment Management menu, allow the specification of codes and commands:

- Query Window Environment** Use a code and command at the *Customize Selection Editor* prompt to specify the action that should occur when the user presses [Shift-F1] while in Query mode. Normally, [Shift-F1] will call the Query Select window.
- General System Configuration** Use a code and command at the *View Mode Sentence Builder* prompt to specify the action that should occur when the user presses [Alt-V] from the View window or from an application window. Normally, [Alt-V] will call the RLIST window.

Codes and Commands in the Make Menu Window

Make Menu, the menu definition window, uses a code and command to define the operation for each menu selection. For example, use the code W and the command INVOICE to display the INVOICE window when this selection is chosen.

Codes and Commands in the Macros Window

Macros, the macro key definition window, uses a code and command to define the operation for each macro key. For example, use the code X and the command EDIT NOTES * to edit all records in the NOTES file.

Codes and Commands in the Secure User and Secure Account Windows

The Management menu provides a path to two windows that use codes and commands: these are the Secure Account window in the Accounts menu and the Secure User window in the Users Management menu.

The Secure Account window allows you to define the characteristics of an account. The Secure User window allows you to define the characteristics of a user. In both cases, codes and commands can be used at the *Restart* prompt.

Use *Restart* to specify an action to perform each time the system goes through its normal restart function. The restart process also occurs when users type END at the debugger or when they explicitly enter the RESET command at TCL. For example, the code M and the command SALES will restart the account at the Sales menu.

Codes and Commands in the Capture Processes

Press [Alt-8] during a capture process to temporarily suspend the capture process. The current keystroke set will display in the full screen Editor, allowing you to add code and command sets into the captured keystroke set. The code and command should be on its own separate line, and should have the format *@code command@*. For example, a code of HT and a command of “The next step shows how to complete an order.” displays the timed message at that point in the playback of the captured keystrokes. This allows commentary to enhance a demonstration procedure.

☞ If you want to use the @@S code within a captured keystroke set, the syntax is:

```
@@@S capture.set.name@
```

The first @ and last @ serve as punctuation to indicate to the keystroke interpreter the beginning and ending of a code and command. They are used any time a code and command is added to a capture set. The @ code itself is doubled to tell the keystroke interpreter that the @ is an executable code rather than an indicator that a code and command sequence is ending.

The Codes and Commands

This section lists the codes and commands that you can use to perform operations in Advanced Revelation. For information about options that can be used with a code and command, see “The Code and Command Options” later in this chapter.

!

Use the ! code when you want to display the date, time and date, or the Advanced Revelation serial number. The command can be the words DATE, TIMEDATE, or SERIAL. This allows you to display this information as a “smart” label in a window. For more information on label display, see the “Adding Advanced Features to Windows with Paint” chapter.

@@S

Use the @@S code when you want to execute a captured keystroke set. The command is required and is the name of the capture set to execute. The capture set is a record in the CAPTURED file. The syntax for the code and command is:

```
@@S capture.set.name
```

{

Use the { code when you want to evaluate a symbolic field in the current dictionary. The command is required and is the name of a dictionary record in the current file. The { code will run the R/BASIC code in the specified dictionary record and return the result. For example, the code { and the command INV_TOTAL might run R/BASIC code to calculate an invoice total and return the result. For more information on using dictionary records, see { }, and CALCULATE in the “R/BASIC Command Reference” chapter, and “Interfacing Dictionaries and R/BASIC” in the “Programming with R/BASIC” chapter in *R/BASIC*.

A

Use the A code to return to the last anchored menu, skipping intermediate menus. There is no command used with the A code. For example, in the Advanced Revelation menu system, to access the Indexing menu, you use two intermediate menus, the Tools menu and the File Tools menu. If desired, the Indexing menu could have included a *Main Menu* option defined with the A code. The option would be used to return directly to the Main menu, skipping over the intermediate menus.

B

Use the B code when you want to use an index to look up the record key for a known data value. If more than one value is found, a popup is displayed. The syntax of the command is:

```
file*i.field {,...} (*d.field {,...} ) (*mode)
```

The four arguments are separated with asterisks (*). *File* refers to the indexed file. *I.field* refers to the field(s) in that file by which you wish to do your index lookup. *D.field* is the name of field(s) that will be displayed along with the record key to allow you to distinguish those record keys.

There are two *modes* that can be specified: KEY or MULTI. If the mode is 'KEY', the indexed records are displayed in a multiselection popup and the returned record keys are placed in a browse list. If the mode is 'MULTI', the indexed records are displayed in a multiselection popup and the returned record keys are @VM delimited for display in a multivalued field. If the mode is null, the record keys are displayed in a single selection popup.

For example, the code B and the command CUSTOMER*COMPANY.XREF*COMPANY,CITY will access the index established for the COMPANY field in the CUSTOMER file. When the index popup displays, the two fields COMPANY and CITY will be displayed along with customer numbers (the record keys for the CUSTOMER file) for the user to choose from. One selection will be returned.

C

Use C to compile and run R/BASIC code. The command is required and is the R/BASIC source code. Each line of code can be separated by an ampersand (&). The command can contain up to 34K of R/BASIC code. For example, the code C and the command SUM(AMOUNT) sums the multivalue field AMOUNT and returns the total. For more information on R/BASIC, see *R/BASIC*.

E

Use the E code when you want to execute a TCL command. The command is required and may be any command used at TCL. The command is typed using the same syntax as at TCL. No active list of record keys is passed to the command. For example, the code E and the command EDIT BP BORDER will edit the program BORDER in the BP file when activated. For more information on the operation of the E code, see the entry on EXECUTE in the “R/BASIC Command Reference” chapter in *R/BASIC*.

F

Use the F code when you want to execute a function key. The command uses a code to indicate which function key to operate. The following are examples of key codes:

- F1 for F1
- SF1 for Shift F1
- CF1 for Ctrl F1
- AF1 for Alt F1

For example, the code F and the command F2 when used as a pre-prompt process will activate the [F2] (Options) key before placing the cursor at the entry line of the prompt.

H

Use the H codes to display help. Each type of H code provides a different way for the information to display. The syntax for the command is determined by the type of H code used. These are the available codes:

- H Use the H code to display help that is stored in the SYS.HELP or the HELP file. The command is a number that is placed at the end of the record key. The record key has this format: *AW*template_name*number*.

Template_name is the name of the window using this help and the *number* is commonly the prompt number for the help. For example, the code H and the command 1 when used for detail help will display the help for the first prompt in the window where it is used.

HD Use the HD code to display help from the *Description* field in a dictionary record. The command is optional. If the command is not filled, the help information is read using the current prompt name as the key to the dictionary record. If a command is used, it contains the name of the dictionary record with the help information. For example, the code HD and the command ST will display the help from the dictionary record ST.

HF Use the HF code to display help from a record or the contents of a field in a record in any specified file. This makes it possible to use a file other than SYS.HELP or HELP to store your help records. The command is required and is typed using one of three syntaxes:

filename, key, field_number

filename, key

key

Filename is the name of an attached file, *key* is the name of the help record, and *field_number* is the number of the field for the help information. For example, the code HF and the command

HELP,CHECK_REGISTER, 4

will display the help information in the fourth field of the CHECK_REGISTER record in the HELP file.

The code HF and the command

PRINTER_HELP,PRE_PRINT

will display the help information in the PRE_PRINT record in the PRINTER_HELP file.

The code HF and the command

PRE_PRINT

will display the help information in the PRE_PRINT record in the SYS.HELP or HELP file.

HL Use the HL code to display a help message that is typed at the command prompt. The command is required and contains the help message. For example, the code HL and the command 'Type a five digit, nine digit, or Canadian zip code.' will display the typed message when the code and command are activated.

HM Use the HM code to display a help message and request a user response. The message is stored in the MESSAGES file or the message text is typed at the *Command* prompt. The command is required and is typed using one of these syntaxes:

key

literal_message_text

The response is returned in the system variable @ANS.

HMD Use the HM code to display a help message that waits for input before leaving. The command is required. It contains the key to a record in the MESSAGES file or it contains the message that is displayed. For example, the code HM and the message 'Make sure the printer is on and ready.' will display the message in a window that displays the message 'Press any key to continue'.

HT Use the HT code to display a timed help message. The command is required and contains the message to be displayed. The default duration of display for timed messages is set in the Message Environment window. To explicitly specify the number of seconds that the message should display, enter ~ (a tilde) followed by the appropriate number. For example, the code HT and the command 'Make sure the printer is on and ready.~10' will display the message for 10 seconds.

K

Use the K code when you want to execute a series of keys. The command uses a code to indicate which keys to operate. The command is filled by pressing [Ctrl - —] (Control + hyphen), which toggles the keystroke translator, then typing the keystrokes required to execute the desired process. When you have finished typing the keystrokes press [Ctrl - —] to toggle the keystroke translator back off. There is a pause before the first keystroke is converted and printed to the command entry line. For example, the code K and the command {CTRL-F3} {CTRL-U} when the editor is on will cut a block to the active buffer and clear the block definition.

L

The L code, like the S code, enables you to call a cataloged R/BASIC subroutine. However, unlike the S code, the L code does not provide access to window registers. The command is required and is the name of a subroutine or the name of a subroutine with an argument. The syntax of the command is:

```
subroutine {,argument}
```

The subroutine must be cataloged. The optional argument must be a value or a literal. For more information, see the S code in this chapter.

M

Use the M code to display a menu. The command is required and contains the name of the menu to display. For example, the code M and the command MAIN will display the Main menu when the code and command are activated.

O

Use the O code as a post-prompt process to enable the user to vary the data files that a window can access. There is no command with the O code. The user specifies what file to use by entering the filename at the prompt. The current file is closed and the file specified by the user is opened for the window.

P

Use the P code to display a popup. The command is required. The command can specify a system popup or a popup record.

Using a Make Popup Record

To specify a popup record defined by the Make Popup window, use the syntax

```
filename*recordname
```

Filename is the file where the popup template is located and *recordname* is the key to the popup template. For example, the code P and the command MEMBER_TYPES displays the popup template MEMBER_TYPES from the POPUPS file when it is activated.

Using a System Popup

To specify an available system popups, use one of these commands:

Use one of these to display a popup with record keys. The @ID popup returns one record key and the @IDS returns one or more record keys from the current file.

Use one of these to display a popup with a list of attached files. The @FILE popup returns one file name and the @FILES popup returns one or more file names.

Use one of these to display a popup with a list of attached volumes. The @VOLUME popup returns one volume name and the @VOLUMES popup returns one or more volume names.

S

Use an S code when you want to call a cataloged R/BASIC subroutine. The command is required and is the name of a subroutine or the name of a subroutine with an argument. The syntax of the command is:

```
subroutine {,argument}
```

The subroutine must be cataloged. The optional argument must be a value, a literal, or the number of a window register. The S code supports the use of <*n*> (where *n* is the number of a register) to pass the contents of a window register. For example, the code S and the command AR_SUB calls the AR_SUB program when the code and command are activated. The code S and the command AR_SUB,<4> calls the AR_SUB program and passes to it the contents of window register 4. For more information on calling subroutines, see CALL in the "R/BASIC Command Reference" chapter in *R/BASIC*.

V

Use the V code when you wish to display a popup of data values from an indexed field. The syntax for the command is:

```
file,field,code
```

File is the name of the indexed file, and *field* is the name of the indexed field. The *code* can be M or S. "M" will display a multiple-selection popup and "S" will display a single-selection popup. For example, a code of V and a command of CUSTOMER,CUSTOMER_TYPE,S will display a single-selection popup of all the customer types already on file.

W

Use the W code when you want to display a window. The command is optional and contains the window name. When the window name is not specified, the window specification window is displayed. Type the window name at the entry line of this window. When the command is the window name, the specified window is displayed. The command option TABLE.ON can be used to display the window in table mode. For example, the code W and the command CUSTOMER TABLE.ON displays the CUSTOMER window in table mode.



X

Use the X code when you want to perform a TCL command that uses an active list of record keys. The command specified may be any command used at TCL. The command is typed using the same syntax as at TCL. An active list of record keys can be passed to the command, or the command can generate an active key list to be passed back to the calling routine. For example, the code X and the command LIST CUSTOMER will list the CUSTOMER file when performed. Any list of record keys that is active before the code and command are performed will be used by the LIST command.

For more information on the operation of the X code, see PERFORM in the "R/BASIC Command Reference" chapter in *R/BASIC*.

The Code and Command Options

This section lists options that can be used in conjunction with any code and command.



;

Use ; (semicolon) to execute a series of codes and commands consecutively. The semicolon follows each code and each command to be executed. For example,

```
S; S; P PROGRAM1; PROGRAM2; POPUPS*INSURANCE.CODES
```

runs three codes and commands in succession. First the subroutine PROGRAM1 executes, followed by PROGRAM2. Finally, the INSURANCE.CODES popup displays.



P

Use the P option with any code to suspend the playback of captured keystrokes before executing the code and command. When the code and command process is completed, playback resumes.

Codes and commands are added to a keystroke set by editing the keystroke record in the CAPTURED file. The code and command should be on its own separate line, and should have the format @code command@. For example,

```
@SP SIMULATE.INPUT@
```

could suspend keystroke playback and run a subroutine called SIMULATE.INPUT that requires the user to enter keystrokes. When the user responds appropriately, execution of the code and command is complete. At that point, keystroke playback will resume.

R

Use the R option with any code to save away the screen image before that code is performed. At the completion of the code and command, the screen image is restored. For example, you might use the R option to restore the screen display following the execution of a code and command that prints to the screen:

```
SR INVOICE.RPT,<1>
```


X

Use the X option with any code to specify that a post prompt process code and command will run only if the data at the prompt changes. For example, the X option might be used in conjunction with an S code and command to run a post prompt process that writes an entry to an audit file each time the value at a prompt changes:

```
SX AUDIT.UPDATE
```



Chapter F 10

Using the Batch Update Process

Advanced Revelation provides a facility for updating a number (a batch) of records in a file in one process. This chapter explains how to set up and use the Batch Update process.

About Batch Updating F 10 . 1

Creating a Batch Update Process F 10 . 1



About Batch Updating

Records in a file are usually changed by accessing them through an application window. The record is retrieved, modified by the user, and rewritten to the file. This process is repeated for each record to be changed.

This manual procedure has certain disadvantages however, which become readily apparent as the number of records to be processed increases. There are three major disadvantages:

- It is slow. Even under ideal conditions, only a few records per minute (per user) may be updated.
- It is labor-intensive. One dedicated operator is required (or more than one operator if the file maintenance is being done in a networking environment).
- It is error-prone. As the user must enter the changes manually for each record, there is a high probability of keyboard entry errors and/or records being missed entirely.

Through the use of the Batch Update process, uniform changes to all (or a selected group) of records in a given file may be made quickly, without tedious manual labor, and with a high degree of reliability. The user simply specifies R/BASIC formulas used to modify the appropriate fields.

Using Batch Update

There are three functions available from the Advanced Revelation Batch Update menu: **Batch**, **Compare** and **Integrate**. These functions are used (respectively), to create and execute a batch process, to compare batch process transactions records with the original file, and to integrate batch process transactions records with the original file. The online help, [F1] (Context) and [Ctrl-F2] (Concept), provides details on the operation of the Batch Update windows.

Creating a Batch Update Process

The first step required to perform a batch file update is to create a batch update definition record. These records define what will happen during the update process, and are created and/or modified with the Batch command window. Definition records are stored in the UP-DATE file.

Using an Intermediate Transaction File

The Batch Update process writes record changes to a special transaction file. the transactions file may then be examined for accuracy, and the changes posted to the data file in a separate step. This approach provides a high level of database protection.

Building the Batch Definition Record

Use the Batch command window to construct the update process definition record.

If only selected records in the file are to be updated, a list of the appropriate record keys should be made active. This may be done with the Filter key [Ctrl-F10]. For more information about the use of key lists, see the “Selecting and Browsing Through Data” chapter in *Accessing Data*. For details about the operation of the Filter window, consult the online help, [F1] (Context) and [Ctrl-F2] (Concept).

The Batch Update process is started from the Batch command window.

Processing Batch Update Transactions

The transactions must be integrated into the original file. This is done with the Integrate command window. The new field values in the transaction file will overwrite the old fields in the original file. When finished with the integration process, the Batch Update process clears the transaction file in preparation for the next batch update. For details about the Integrate window, consult the online help, [F1] (Context) and [Ctrl-F2] (Concept).

Before integration, you can examine the contents of the transaction file. To do so, use the Compare window. For information about the operation of the Compare window, refer to the online help, [F1] (Context) and [Ctrl-F2] (Concept).

Examples of Batch Updating

The following two examples illustrate the use of the Batch Update process:

- An employer wishes to give a \$100 payroll bonus to all employees who took no sick leave. This could be accomplished in 2 steps. First, he would use **Select** on the Filter Tools menu to obtain a list of record keys with the DAYS_SICK field equal to zero. Second, he would enter and execute a batch process definition record that adds \$100 to the BONUS field for these employees.

- A company wishes to add a daily compound interest amount to all overdue invoices. A batch update definition would be created to calculate the amount of interest payable on the PAST_DUE amount field and add it back into this field.

Since this interest charge process is to be run daily, it should be made a TCL batch command. Using the Link Multiple Command window, this update process might be given the name CHARGE_INTEREST. Once saved under this name, the update could be run at any time by simply typing CHARGE_INTEREST from the TCL window. For more information see the “Introduction to TCL” in *TCL*.

CHARGE_INTEREST could also be run from an R/BASIC program (see PERFORM and SELECT in the “TCL Command Reference” chapter in *TCL*), or as an option on a menu (see the “Building Menus and Popups” chapter).



Chapter F 11

Writing Macros and Capturing Keystrokes

This chapter explains how to define and use customized sets of macro keys to simplify and speed up operations and procedures in Advanced Revelation. It also describes the Capture utility which is used to capture and replay keystroke sequences. The Capture utility enables you to produce self-running demos and tutorials for your Advanced Revelation applications.

Defining and Using Macro Sets	F 11 . 1
Capturing Keystrokes	F 11 . 3



Defining and Using Macro Sets

The keys [Alt-1] through [Alt-5] have been reserved for use as macro keys. Each of these keys can be programmed so that it performs a special operation any time it is pressed. Any TCL or R/BASIC command can be assigned to a macro key, placing often used commands at your fingertips.

When to Define Macro Keys

Macro keys provide operating shortcuts. If you frequently type a particular command at the TCL window, consider defining a macro key to issue the command for you. Define and use macro keys to streamline the operation of activities that you do often. They make repetition of an operation faster and more convenient.

If a more complicated procedure involving many operations will be performed routinely, use the options on the Capture menu to record and duplicate it.

Creating a Macro Set

The operations you attach to the five macro keys are stored in a macro set. Give the macro set an identifying name. Assign a code and command to each key in the set to define the operations that the keys will perform. Press [F9] to save the information contained in the macro set.

Each macro set that you define and save becomes a record in the MACROS file. You can define as many macro sets as you like, although only one macro set can be active at a time.

Environment

The ability to build and execute macros depends on information maintained in the General System Configuration window, which is accessed by using the **Management-Environment-General** options from the Advanced Revelation Main menu. Use the General System Configuration window to activate macro building, to activate macro execution, and to determine which macro set is active when you log on to Advanced Revelation.

Accessing the Macros Window

If macro building is activated, the Macros window can be called from any place within Advanced Revelation by pressing [Alt-M]. To access the Macros window from the menu, select **Design-Macros**.

When the Macros window appears, it displays the active macro set, identifying it by name. At this point you can do a number of things:

- Use the Macros window to create a new macro set
- Add to or edit the key functions in the active macro set
- Display another macro set, making it the active set

Defining a Macro Key

Each macro key is defined by means of a code and command. Code and command options are explained in the “Using Codes and Commands” chapter. Below are some of the codes used frequently for macro keys.

When you want to:	Choose code:
Execute a command that will use or create a record key list	X
Execute a command that will not use or create a record key list	E
Reproduce the keystrokes used to carry out an operation	K
Display a window	W
Display a popup	P

Using Codes and Commands

Use codes and commands to define macro keys that issue particular commands at the TCL window. Define the macro key with a code (for example, X for System Perform, E for System Execute) and then type the command that should be carried out when the key is pressed.

When the macro key is pressed it suspends all processing to execute another subsystem or process. It sends the specified command to The Command Level (TCL) processor for execution at the next level, and then returns to the calling level.

Using the Keystroke Translator to Define a Macro Key

You can program a macro key to reproduce a short series of keystrokes that control some operation. Use code K (Macro Keystrokes). After entering K at the *Code* prompt, tell Advanced Revelation the names of the keystrokes in the macro by using the keystroke translator.

The Keystroke Translator

[Ctrl - —] is a toggle switch that turns the keystroke translator on and off. Type [Ctrl - —] to turn on the translator. There will be a delay while the translator is loaded into memory. Wait a few moments, then type the sequence of keystrokes that you want to reproduce whenever the macro key is pressed. When the translator is on, it processes each keystroke and writes out the name of each key in a format that Advanced Revelation recognizes. The keystroke translator remains active until it is toggled off by pressing [Ctrl - —] again.

Note that you can alternately use the Capture utility to record and view the keystrokes that should be entered at the *Macro Keystroke Command* prompt. When a longer keystroke sequence will be performed routinely, use the Capture utility to record and play it back. For more information on recording, viewing, and playing back a keystroke sequence, see “Capturing Keystrokes” later in this chapter.

Making a Macro Set Active

To change the active set through the Macros window, type in the name of the macro set that you want to activate and press [Enter]. Advanced Revelation will read the MACROS file and display the requested record. As soon as the record is displayed, it becomes the active macro set.

Another way to change the active set is to use the MACRO command at the Command Level (TCL). Use the following syntax:

```
MACRO macro.name
```

As soon as the command is entered, the named set becomes active.

Capturing Keystrokes

The Capture utility enables you to record and play back long keystroke sequences. This makes it possible to duplicate any procedure. If you perform an activity routinely, you can save execution time and simplify operation by recording the procedure in a capture set. Once captured, the keystroke sequence in the capture set can be played back on demand.

When to Use Keystroke Capture

The Capture utility records the sequence of keys that are pressed to perform a procedure. This set of keystrokes, played back in the proper context, will reproduce all of the operations in the procedure.

If a complicated procedure will be performed routinely, use the Capture utility to duplicate it. When only a few keystrokes need to be reproduced, use the Macro Keystrokes code and command options. For more information on Macro Keystrokes, see “Defining and Using Macro Sets” earlier in this chapter.

In addition, keystroke capture allows you to create self-running demos and tutorials for Advanced Revelation applications. An entire procedure can be run, stopped at key points, and then continued.

Finally, since the Capture utility records the keystrokes used to perform an operation, it can serve as a reference when you produce documentation for an application, ensuring that all steps in the process are covered.

A capture set can be simple or complex. You can create individual capture sets or you can link capture sets by using the Editor, or by specifying code and command options.

When capture sets are linked to one another, they can produce a more complex result like a self-running demo program or a tutorial. You must use more advanced techniques to link and modify capture sets. This may include writing a program to link a series of keystroke sets, or using the Editor to merge one capture set into another.

- ☛ It is important to remember that playing a sequence of keystrokes produces useful results only if it is started under the same conditions and at the same point in Advanced Revelation where the initial capture began. This can be controlled by starting keystroke capture and playback with [F5] (TCL window), then [F10] (Menu).

Environment

The ability to capture sets of keystrokes depends on information maintained in the General System Configuration window, which is accessed by using the **Management-Environment-General** options from the Advanced Revelation Main menu. Use the General System Configuration window to activate the Capture utility and to set keystroke playback delay. If Capture is not activated, you will not be able to start capturing keystrokes but you will be able to play back existing captured sets.

Running the Capture Utility

To run the Capture utility from the Advanced Revelation Main menu, select **Tools-Capture**, and then choose the option you want to use.

You will find that it is more practical to use the special macro keys to run most of the Capture options. This avoids capturing the keystrokes that must be used to exit and access the Tools menu and to select the **End Capture** option. It also allows you to control more accurately the point at which keystroke playback begins.

Keystroke Capture Operations

When you want to:	Press:	or Choose menu option:
Start capturing keystrokes	[Alt-0]	Start
End capturing keystrokes	[Alt-0]	End
Suspend a capture sequence	[Alt-8]	
Play a captured set of keys	[Alt-9]	Play
Modify a captured set of keys		Modify
Stop capturing keys without keeping them		Cancel

Note that the **Modify** and **Cancel** options are only available from the Capture menu. The [Alt-8] function, which allows you to suspend a capture process so that you can use the Editor to make additions and modifications to the keystroke sequence, is not available from the Capture menu.

Creating a Capture Set

You can use [Alt-0] as a toggle switch to turn the keystroke capture process on and off. Type [Alt-0] at any point in Advanced Revelation to start the capture process. This is the quickest way to capture a keystroke sequence. Alternatively, choose **Start** from the Capture menu: the Start Capture window appears.

You are asked to name the capture set that will be created. Capture sets are stored as records in the CAPTURED file. You can press [F2] to see a list of existing capture sets. These may be reused. Type a new name or select an existing set name and press [Enter]. Capture begins to record every keystroke that follows.

- ☛ Part of your captured keystroke sequence may involve making selections from menus. When capturing selections off a menu, always type the first character of the menu option and press [Enter] to select the item. This ensures that when the capture set is played back, the proper menu selections will be made.

Never move the cursor to make a menu selection, as the resulting capture set may play back differently each time it is run. If you do, the menu defaults that are active during keystroke playback can divert the keystroke sequence to a different path, resulting in meaningless processing. This is because the active default menu options at the time of playback vary, based on earlier menu selections made during the Advanced Revelation session. The starting point (default menu option) from which the cursor will move in a menu is not predictable.

Ending Keystroke Capture

Once capture is initiated, each keystroke will be recorded, in order of occurrence, until keystroke capture is turned off. When creating a simple capture set, use [Alt-0] to stop the capture process or choose **End** from the Capture menu. You will see the messages:

`record.name` is being saved in the CAPTURED file.

Converting keystrokes to literal format.

At this point, the captured set can be played back on demand or modified.

Suspending the Capture Process

Use [Alt-8] to temporarily suspend the capture process. The current keystroke set will display in the full screen Editor, allowing you to add to and edit the set.

Editing Text You can edit the captured keystroke sequence using the Editor. This permits you to remove or change keystrokes that you might have made in error.

Adding text You can also add text into the captured keystroke set. By adding text, you can make use of features of the captured keystroke process that you could not otherwise access.

You can add comments to the captured keystroke set. This enables you to insert remarks or notes into your captured keystroke sequence for later reference. An asterisk is used to identify a comment. When an asterisk is entered, all text that follows it to the end of the line is treated as a comment, and will be ignored by the captured keystroke replay process.

You can also add code and command sets into the captured keystroke set. The code and command should be on its own separate line, and should have the format `@code command@`.

While in the Editor, you can use the keystroke translator to add keystrokes to a capture set. For more information, see “Adding Keystrokes to a Capture Set” later in the chapter.

Once you are finished editing the captured keystroke sequence, press [F9] to save your edits. Then press [Esc] to resume keystroke capture. You will return to the same point at which [Alt-8] was pressed. To terminate capture at that point press [Alt-0]; otherwise keystroke capture continues.

Canceling the Capture Process

To cancel keystroke capture without saving the keystroke sequence, choose **Cancel** from the Capture menu. The following message is displayed:

Capture Process canceled

Playing Back a Capture Set

A capture set can be replayed as often as needed. Be sure to start replay under the same conditions and at the same point in Advanced Revelation where the initial capture began.

To play back a set of keystrokes from any point, press [Alt-9]. Alternatively, if the context is appropriate, choose **Play** from the Capture menu. The Play Capture window appears.

You are asked to name the capture set to play. You can press [F2] to see a list of existing capture sets. Type the set name or select a set name from the list and press [Enter]. Whatever keystrokes were captured in the named set will be replayed immediately.

Viewing and Modifying the Keystrokes in a Capture Set

Since the keystrokes in a capture set perform some procedure or series of operations, you can add or omit keystrokes to vary the actions that will occur.

To see a list of the captured keystrokes, and to make modifications, choose the **Modify** option on the Capture menu. The Modify Capture window appears.

You are asked to name the capture set to modify. You can press [F2] to see a list of existing capture sets. Type the set name or select a set name from the list and press [Enter]. The Editor window appears, displaying the keystrokes that were captured.

You can edit the captured keys, deleting key names where necessary. If you want to add keystrokes to the set, use the keystroke translator to assist you. When editing is complete, press [F9] to save the changes. To abandon your edit and preserve the capture set in its original form, press [Esc].

For advanced users who want to reproduce a complex procedure, the full screen Editor makes it possible to merge capture sets into one another.

Adding Keystrokes to a Capture Set

You can add keystrokes to a capture set by using the keystroke translator. When using the translator, as you press the keys, the translator types the names of those keys in a format that Advanced Revelation can recognize.

Using the Keystroke Translator

[Ctrl - —] is a toggle switch that turns the keystroke translator on and off. Position the cursor at the place where you want to insert keystrokes and type [Ctrl - —] to turn on the translator. There will be a delay while the translator is loaded into memory. Wait a few moments, then type the sequence of keystrokes that you want to add. When the translator is on, it processes each keystroke and writes out the name of each key in the format that Advanced Revelation recognizes. The keystroke translator remains active until it is toggled off by pressing [Ctrl - —] again.

- ✎ Even keys such as [Spacebar] or [Enter] or [Esc] are converted by the translator. For example, when you press [Tab] the translator writes out {TAB}.

Chapter G 1

Selecting and Browsing Through Data

This chapter provides an overview of how Advanced Revelation uses lists of record keys to locate the records used in a program or command. It also explains how record key lists are used in an application window to view specific records from a data file.

Using Record Keys	G 1 . 1
Defining Selection Criteria for Record Key Lists	G 1 . 3
Using Browse Lists in Application Windows	G 1 . 4



Using Record Keys

Every record in an Advanced Revelation file has a unique identifier. This unique identifier is called the record key.

Record Keys

In order to access a record, Advanced Revelation uses the record key to determine the record's location. You must provide the record key for each record you want to view or process.

The record key consists of a string of alphanumeric characters that may also contain \$. - _ . A record can have only one record key, and a record key can refer to only one record.

For example, you might identify the customers in your CUSTOMERS file by assigning to each one a unique customer number. When you wish to access a specific customer record, you give Advanced Revelation the customer number of the record you require.

Many processes in Advanced Revelation (for example, windows) ask you to provide a key so that they can find the appropriate record to process.

Record Key Lists

Sometimes you will want to access not just one record, but a series of them. To do this, you can use a list of record keys.

For example, you may wish to send a mailing to all of your customers who reside in the state of New York. Rather than having to provide each individual record key to Advanced Revelation, you can have Advanced Revelation build and use a list of record keys corresponding to the records of the appropriate customers.

Once you have a record key list, Advanced Revelation can pick out each individual record key from the list, and then process the record represented by that key.

Using Lists of Record Keys

You will be making use of record key lists in a variety of places. Some examples:

- Reports** You might create a list of record keys that represents a subset of the data in a file. This list can then be used when you produce a report with R/LIST, or with the Form or Label processes.

Collector windows You might use a list of record keys when processing information in a collector window. For example, you might want to copy selected records from one file to another. In that case, you would use a list of the appropriate record keys in the Recordcopy window.

Application windows You may wish to view or update a selected group of records in an application window. To do so, you can create a record key list, and then use the list to call each record into the window.

Creating Lists of Record Keys

There are three ways to create lists of record keys. In some cases, you will be able to enter the list yourself when prompted by a process in Advanced Revelation. This might be the case if you were using a collector window such as that used by the Edit or Recordcopy process.

Many times, however, you will not know all the record keys to be processed, or will find it too tedious to enter them all. In those cases, Advanced Revelation can generate a record key list for you.

Advanced Revelation uses a process called filtering to build a list of record keys. To filter, Advanced Revelation looks through each record of the file being used (or through the appropriate index) and determines whether the record meets criteria you specify. If so, the record's key is added to the list. When Advanced Revelation has finished with the file, the list of record keys is made available to you.

Finally, a special list of record keys is maintained in application windows. As an operator edits records using a window, Advanced Revelation keeps track of the record keys for those records. This list is available at any time by using a special keystroke [Alt-U] in the application window.

Types of Record Key Lists

Once a record key list has been created, Advanced Revelation makes it available to you in one of two ways.

Select Lists In a collector window or at TCL (The Command Level), Advanced Revelation makes the list available as a *select list*. When a select list is available, it is said to be "active," and the word "Filter" will appear in the status line, since the records in the file have been "filtered".

In most cases, the next process you execute in Advanced Revelation will use only the records *represented by the* currently active select list. As the report or command uses a select list, each key is used and then discarded. When no keys remain on the select list, the report or process is complete.

An active record key list is available for the next process only. If the list isn't used by that process, it is cleared. If you wish to use the active list for several processes, or if you wish to use it in a later process, it can be given a name and saved in the LISTS file. Any saved list can be retrieved and used in other processes.

Browse Lists

In an application window, the record key list operates as a *browse list*. Typically, the browse list is used to scan ("browse") through a series of records in order to locate or view the exact record you want. Unlike a select list, the keys on a browse list are not used up. Instead, the browse list operates in a circular fashion. You can page through the records named on the list, moving forward or back. When you reach the end of the browse list, you circle back to the beginning and continue through again.

Unlike a select list, when a browse list is created you have an opportunity to edit it before using it.

- ☛ The status line notes whether the current window is an application window or a collector window. The word "Collector" displays when you are in a collector window. When you are in an application window, the status line shows the name of the file.

Defining Selection Criteria for Record Key Lists

When you want Advanced Revelation to create a record key list, you must specify criteria to define which records in the file will qualify for your list. The selection criteria is specified in a SELECT statement.

A SELECT statement tells Advanced Revelation to select records from a file for use in a process or report. The sentence contains the selection criteria used to create a record key list that names the records to include in the process.

Entering SELECT Commands

You can enter and execute a SELECT command in several ways. A SELECT sentence can be entered at the TCL level, or at the Filter or Select windows (which are accessed from the Filter Tools menu). Within data entry windows, you can use Advanced Revelation's Query facility or you can use the Filter key [Ctrl-F10].

The Query facility allows you to enter selection and sort criteria at the prompts in a data entry window. Query uses the information you enter to construct and execute a SELECT command. For more information about Query, see the "Using Query" chapter.

The Filter key presents you with several ways to create or activate a selected list of record keys, such as using the Filter or Select window.

The Filter window allows you to save selection criteria as a Filter you can reuse. This is useful when you expect the selection criteria to remain static (that is, you will use the same criteria repeatedly against data in that file), but anticipate that data in the file will change often.

The Select window allows you to set up criteria to be used immediately and then discarded.

EasyWriter is available in both the Filter and Select windows to help you compose a SELECT statement. While in the Select or Filter window, press [F2] (Options) to call EasyWriter. Use of this simple-to-operate, popup-driven tool is explained in the “Writing Reports with EasyWriter” chapter in *Creating Reports*.

Details about the options available from the Filter key [Ctrl-F10] follow this section.

For more information about setting up selection criteria with a SELECT statement, see the SELECT entry in the “TCL Command Reference” chapter in the *TCL* section. Also see the chapter on creating R/LIST reports in *Creating Reports*.

Using Browse Lists in Application Windows

If you are in an application window you can generate an active list of record keys using the **Select**, **Filter**, or **Query Table** options. You can also have Advanced Revelation provide you with a list of record keys for changed records by pressing the [Alt-U] key.

Overview of Functions

The [Alt-U] key creates a browse list out of all records that have been entered or changed during the current session. This permits you to examine and print (using [Alt-P] — see below) changed records.

The list of keys is initially put into a Browse Edit window. Using the Browse Edit window, you can add and delete keys as desired. When you finish review or editing, press the Save key [F9] to make the browse list active. For details about the Browse Edit window, see below.

When a browse list is active, the word “Browse” appears on the status line. Press [Alt-F] to move forward to the next record. Press [Alt-B] to move back to the previous record. Browsing through records on a browse list is circular. When you reach the end of the list, Browse cycles you through to the beginning again. Similarly, paging back through the first record on the list will place you at the end of the record list.

You can accomplish other functions while using an active browse list. If you wish to edit the browse list, you can press [Alt-I], and the Browse Edit window will appear. You can edit keys in this list normally.

You can also use [Alt-P] to print all the records in the browse list. If you press [Alt-P], Advanced Revelation will use the current window as a form template, and print each record on the browse list directly to your printer.

Summary of Browse List Keystrokes

Key	Function
[Alt-I]	Bring up the Browse Edit window
[Alt-F]	Browse forward one record
[Alt-B]	Browse backward one record
[Alt-P]	Print the current list to the printer, using the current form as a template
[Alt-U]	Create a browse list of records changed during the current session

The Browse Edit Window

In an application window, you can display the Browse Edit window at any time by pressing [Alt-I]. The record keys on the active browse list are made available for editing. If no list is active, you can create one by entering record key names. Press [F9] to make the list active.

Browse Edit Softkeys

Because the Browse Edit window displays only record keys, it has two softkeys to make it easier for you to edit the browse list. Use the softkeys key [F6] to access each of these softkey options, or execute the softkey directly by pressing the appropriate keystroke.

Choosing and Ordering Browse Keys

[Shift-F1] brings up a multiple-choice popup that makes it easier for you to choose specific keys to include on the popup, and to arrange the keys into a specific order. This is useful when you wish to see only a few keys out of many, and when you wish to see the records in an order different from the way they appear in the Browse Edit window.

To access this function, press [Shift-F1] at the Browse Edit window. You will see a multiple choice popup with the first page of record keys from the browse list.

To select keys from the Browse List Select popup, move the highlight bar to the key you wish to include, and press [Enter]. Move the highlight to the next key to include, press [Enter] again, etc. When you are through choosing keys to include on the browse list, press [F9] (Save), which will return you to the Browse Edit window.

When using this popup, you should choose record keys in the order you wish them to appear in the browse list. The first key you choose will be first on the list, the second key will appear second, etc. A number in the right-hand column of this popup tells you the order in which the keys will appear.

Viewing Additional Data for Browse Keys

[Shift-F2] makes it possible for you to see additional information about each record on the browse list. This is handy when you find it difficult to distinguish the records on the browse key list by their record keys alone.

To display the Display Fields Select popup, press [Shift-F2] at the Browse Edit window.

For example, your browse list may contain record keys for records in the CUSTOMERS file. Since you might not know your customers by customer number, you may decide to display the company name and city fields from each record to help you in distinguishing them.

When the Display Fields Select popup appears, it gives you a choice of all fields defined for that file. This is a multiple-choice popup — highlight and select each field to display in turn. When you are through selecting fields, press [F9] (Save).

When you save the fields to be used for browse list display, you will see an expanded version of the Browse List Select popup. This is a list of all keys in the browse list, but it also displays the data in the fields you just chose. This is an ordered multiple-choice popup. Select the records you wish to have on the browse list, as described in the previous section. Save the list with [F9]. This will return the keys you chose to the Browse Edit window.

Passing Select Lists to a Window

If a select list (filter) is active when a window is invoked, the select list will be passed to the window as a browse list. This makes it possible to pre-select data (for example, using a SELECT or GETLIST command), and then feed the data into a window for review.

This capability is typically used when invoking a window from TCL (The Command Level) or from within an R/BASIC program. For example, if you wish to examine all records for customers in the state of Kansas, you could type the following two commands in succession at TCL:

```
SELECT CUSTOMER WITH ST = "KS"
```

```
WINDOW CUSTOMER_ENTRY
```


The first command (SELECT) will create a list of all records in the CUSTOMER file in which the ST field or index contains the abbreviation KS. Once these records have been found, the window CUSTOMER_ENTRY is invoked. The records found by the SELECT command will become a browse list in the window.

- ✎ Although a select list (filter) can be any length, a browse list is limited to a length of approximately 32K (32,768 bytes). If you create a very large select list and pass it to a window, the window will truncate the select list to approximately 32K before turning it into a browse list.



Chapter G 2

 Using the Filter Key

This chapter introduces the Filter key and explains how to use the options available through this key to manage lists of record keys.

Using the Filter Key G 2 . 1





Using the Filter Key

To create and manage record key lists within any collector or application window, press the Filter key [Ctrl-F10]. This produces a popup of selection options.

Depending on where you are in Advanced Revelation when you press the Filter key, you will see slightly different options for creating record key lists. If you are in a collector window or application window for an unindexed file, you will see the Filter Selections popup. If you are in an application window for an indexed file, you will see the Browse Options popup. Both popups are similar, but the Browse Options popup has additional capabilities.

Filter Key Options — Filter Selections

When you press [Ctrl-F10] (Filter) in a collector window or in an application window that accesses an unindexed file, Advanced Revelation displays the Filter Selections popup. The various popup options present you with different methods of creating lists of record keys.

If you are in a collector window, any key list created or activated using this popup operates as the active select list for the current window. If you are in an application window, the selected list of record keys will function as a browse list. For details on using a browse list, see “Using Browse Lists in Application Windows” in the “Selecting and Browsing through Data” chapter.

These options appear on the Filter key popup for Filter Selections:

- Select
- Query Table
- GetList
- SaveList
- Filter

Each option on the Selections popup is described below.

Select Choose the **Select** option to access the Select window. The window assists you in creating a SELECT sentence. The SELECT sentence defines the selection criteria that control which record keys are included on the key list. A record of the selection criteria is not saved permanently. It is stored temporarily in the Query Table, but is eventually discarded.

Once you have finished specifying your selection criteria using this window, use the Save key [F9] to execute the SELECT command.

For information on how to specify selection criteria using the Select window, see the “Writing Reports with EasyWriter” chapter in *Creating Reports*.

Query Table

Choose the **Query Table** option to access the Stored Query Table. This table lists the most recent SELECT commands executed by Advanced Revelation. It provides a convenient way to retrieve the list of record keys generated by any of the SELECT statements in the table. Choose the desired SELECT statement and press [F9].

You can control how many lists are stored in the Query Table by editing the *Number of saved queries* prompt in the General System Configuration window. To access this window, choose the **Management-Environment-General** options from the Advanced Revelation Main menu.

GetList

Choose the **GetList** option to access the GetList window. Use it to retrieve a list of record keys stored earlier in the LISTS file with a Savelist command. Enter the name of the list you want to use and press [F9]. The system will display a message indicating how many records are in the retrieved list. The list is then handled exactly as if it had just been generated.

SaveList

The **SaveList** option allows you to save a record key list so that it can be used at a later time. When you choose this option, the SaveList window appears. Give the currently active list an identifying name and press [F9]. The list is stored as a record on the LISTS file.

A saved list can be used over and over by reactivating it with the GETLIST command. This is useful if you wish to run several reports or processes using the same list of record keys.

☛ The command or process that uses a select list always uses up the list. Therefore, if you intend to save the list, you must use the **SaveList** option before applying the select list to any process.

Filter

Choose the **Filter** option to access the Filter window. A Filter is a SELECT sentence that is saved so that it can be used again.

The Filter window is used to read and execute a Filter record that already exists, or to create a new Filter record. The Filter window writes a record containing your selection criteria to the FILTERS file. The record key list is created when the existing Filter is executed or when a new Filter is saved and executed.

For information on how to specify selection criteria using the Filter window, see the “Writing Reports with EasyWriter” chapter in *Creating Reports*.

Percentage Done and Status Bar

The status line keeps you advised about the progress of your filter operation. As Advanced Revelation works on filtering your data, it prints two messages at the top of the status line.

The first of these is a Percentage Done number. This number is updated as the filter process executes, providing you with a clear indication of how far along the operation is.

In addition to a Percentage Done number, you will also see a graphic Status Bar. The Status Bar is a bar graph representation of the number in Percentage Done. As the filter operation proceeds, the Status Bar grows to the right. When it reaches the right margin of the status line, the filter operation is complete.

- ✎ Under most circumstances, the Percentage Done counter and the Status Bar are only available in files created under version 1.1 or higher of Advanced Revelation.

Conditions for Displaying Percentage Done

As long as the status line is active, the filter process will generally create a Percentage Done counter and the Status Bar. Certain conditions will prevent the Percentage Done counter and Status Bar from displaying. Generally speaking, these two messages are not displayed if Advanced Revelation determines that the filter operation will not last very long. Among the conditions that prevent the messages from displaying are these:

- A report is displaying in the View window.
- The file contains fewer than 100 records.
- The filter is based on one or more indexed fields.
- The filter is not based on a file created under version 1.1 or higher of Advanced Revelation, and this is a new filter (not working against an active select list).

If the status line has been turned off completely, the messages will not appear.

Filter Key Options — Browse Selections

When the Filter key is pressed in an application window for an indexed file, Advanced Revelation produces a popup labeled Browse Options. Any key list created or activated using this popup operates as a browse list within the application window.

The Browse Options popup enables you to execute the same options as the Filter Selection popup discussed above, but gives you the additional capability of creating record lists out of indexed fields.

These options appear on the Browse Options popup:

- Indexed Fields
- Browse Selections
- Use the first option in the popup to see which fields in the file are indexed and to select records based on an indexed field.

Use the second option to create a record key list from unindexed fields. Option 2 calls the Browse Selections popup menu, which is identical to the Filter Selections popup described above.

The Indexed Fields Option

If you choose the **Indexed Fields** option from the Browse Options popup, you will be presented with the Index Selection window, which contains prompts for each indexed field in the current file. If a field has been indexed using a Cross Reference index, the word “XREF” will be appended to the field name to alert you to this fact.

To create a record key list out of an indexed field, move the cursor to the field on which you wish to base your selection, and fill in the value you want to search for in the field. You can specify more than one value, and Advanced Revelation will find the records containing any of the values you enter.

- ☛ You can search indexes not just for exact matches of data, but for substrings, ranges, and specific patterns. These index search capabilities, and the index lookup syntax are described in the “Using Indexes” chapter.

For example, assume you were working in a window containing a field called “STATE”, which had been indexed. You wish to find all records in which the state field contained either “CA” or “AZ”. You would call the Browse Options popup using the Filter key [Ctrl-F10], then choose the **Index Selection** option, which would present you with the Index Selection window.

Move the cursor to the STATE prompt of the Index selection window, and type in the word "CA" and then the word "AZ". When you save your choice with the Save key [F9], Advanced Revelation will search the index and create a browse list out of all the record keys associated with records containing either "CA" or "AZ" in the state field.

If Advanced Revelation finds just one record matching your lookup criteria, it will fill the record into the window. However, if there is more than one record corresponding to your lookup criteria, it will present you with the Browse Edit window. Use of browse lists and the Browse Edit window is explained in the "Selecting and Browsing through Data" chapter.

The Browse Selections Option

The **Browse Selections** option of the Browse Options popup calls up a popup of methods by which you can create record key lists. This Browse Selections popup is identical to the Filter Selections popup. Refer to the discussion above for details on how to use this popup.



Chapter G 3

Using Indexes

This chapter explains the index search options that are available in application windows. For information about establishing indexes, see the “Creating and Maintaining Indexes” chapter in *Developing an Application*.

Introducing Advanced Revelation’s Indexing System . . . G 3 . 1

Index Search Options in Application Windows G 3 . 1



Introducing Advanced Revelation's Indexing System

An Advanced Revelation index is a specially organized list of record keys. When you supply a value to the index, it returns the key or keys to records that contain that value. Indexed fields provide alternative ways to look up the information that is stored in your database. The indexes give you quick, flexible access to any record, and speed up the process of selecting and sorting records.

Finding Information in a Data File

In Advanced Revelation, when you define the record structure for a data file, one field is identified as the record key. This field is used to assign a name to each record in the file. The record key is unique to each record and refers to one specific record in a file.

Records in an Advanced Revelation data file are located by using the record key. When you provide the key, the system can find and display the record.

For example, CUST_NO might be used as the record key field in the CUSTOMER file. Each customer is assigned a customer number, which becomes the key to that record. In the CUSTOMER application window, if you enter the customer number, Advanced Revelation reads and displays the customer's record.

But what if you can't remember the customer number? The problem then is how to provide the record key when you want to read a particular record. An indexing system offers the solution. If you know other information from other fields in the record, such as a company name, or a contact name (both of which are indexed), Advanced Revelation's indexing system will find the key.

Index Search Options in Application Windows

If you have established indexes for your data files, you can make use of these indexes when looking up data in windows.

- You can use \ (backslash) plus a search value directly at the key prompt. This creates a direct index search.
- You can search indexes not just for exact matches of data, but for substrings, ranges, and specific patterns. The Browse Select window, accessible from the Filter key [Ctrl-F10], prompts you for information used to search the indexes.

Records containing the information that you want are listed on browse lists. Once a browse list is active, it can be used in the window to view the records. For more information on using browse lists, see the "Selecting and Browsing through Data" chapter.

Direct Index Lookup (\)

You can initiate an index lookup in an application window directly from the key prompt. This saves you the trouble of first pressing [Ctrl-F10] (Filter), choosing the **Indexed Fields** option from the resulting popup, and filling in data at the Browse Cross Reference window.

To initiate a direct index lookup in an application window, enter \ (backslash) plus the data to look up at a key prompt. For instance, to create an index lookup based on the word ACME, you would enter the following at the key prompt in a window (assume that *Customer No* is the key prompt):

Customer No \ACME

If more than one field is indexed in the file, you will see a popup of all indexed fields. Choose the field for which you are searching. For example, if you wish to look up the value ACME in the COMPANY_NAME field, choose this field name from the popup. If only one field is indexed in the file, no popup of field names appears.

Any record keys found by the index lookup are returned to the application window as a browse list.

Index Lookup Syntax

You can create browse lists in application windows using comparison, substring, range and pattern match searches. This means that you can look up data from the indexes if you know only part of a word that was indexed. You can also create a search based on a range of values, or on a pattern that describes the data.

These search capabilities are available anytime you use the Browse Cross Reference or Shared Index Information windows to search an index. This type of index search technique is often used in application windows. Successful index lookups return a browse list to the application window.

There are three areas in Advanced Revelation that are affected by this index search capability:

- (Ctrl-F10)-Indexed Fields** This Filter key and popup combination, called from an application window, displays the Browse Cross Reference window, enabling you to enter index search criteria. For details on using [Ctrl-F10] with indexes to create a browse list, see “Filter Key Options — Browse Selections” in the “Using the Filter Key” chapter.
- \ (Backslash)** A backslash followed by data at the key prompt of an application window works much like the [Ctrl-F10] **Indexed Fields** option. However, the backslash initiates an index lookup directly, bypassing the Browse Cross Reference window. This capability is described under “Direct Index Lookup” earlier in this section.

Code and Command The “B” code and command sequence displays a Shared Index Information window under developer-specified circumstances (for example, using the Options key). This window prompts you to enter index search criteria. For more information on the “B” code and command, see the “Using Codes and Commands” chapter in *Developing an Application*.

Using Index Lookup Syntax

When accessing indexes and creating browse lists in one of these ways, you can use special characters to create substring, range, or pattern match searches. The index search methods available are listed in the following table:

Table G 3-1

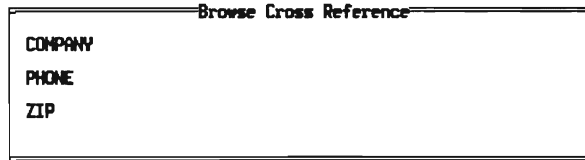
Index Search Methods

Operation	Method	Operator	Example
Comparison	Equal	=	=50
	Greater than	>	>99
	Less than	<	<100
	Greater than or equal	>=	>=100
	Less than or equal	<=	<=1000
Substring	Beginning with	...]	ACM]
	Ending with	[...]	[ING
	Containing	[...]	[TON]
Range	From-To	...	25...50
Pattern Match	Matching	%	%3N?.'4N
Logical	Not	#	#>1000 (not greater than 1000)

Search Examples

Imagine that you have indexed the fields COMPANY, ZIP, and PHONE in the CUSTOMER file. You are at an application window that you use to enter and view customer data.

When you press [Ctrl-F10] (Filter) to access the Browse Selection popup, your first option will be **Indexed Fields**. If you choose this option, a window will appear with the field names in it, similar to this:



Browse Cross Reference

COMPANY
PHONE
ZIP

If you wish to look up all companies in which a portion of the company name is ACME, you would fill in the *COMPANY* prompt like this:

COMPANY [ACME]

This creates a search for all companies in which the string ACME appears anywhere in the name. If you were only interested in companies whose name began specifically with the word NORTH, you would fill in the prompt like this:

COMPANY NORTH]

In another example, if you wish to look at all customers within a certain ZIP code range, you might fill in the window like this:

ZIP 90000...99999

This creates a search for all customers whose ZIP code falls in the range 90000 to 99999, inclusive.

Finally, you might wish to find all customers outside your local telephone area. You can identify these customers easily, since you entered no area code for local customers. You might fill in the window like this:

PHONE %3N' - ' 3N' - ' 4N

This creates a search for all customers whose telephone number consists of three numbers, a dash, three numbers, another dash, and four numbers. In other words, you are looking for customers whose phone number matches the pattern of area code plus number.

☛ For more information on creating match searches, see "Data Validation" in the "Adding Advanced Features to Windows with Paint" chapter in *Developing an Application*.

Searching with Multiple Criteria (AND and OR)

If you create a search using multiple criteria, Advanced Revelation will make certain assumptions about how to construct your query.

If you specify multiple search values for a single field, the query will assume a logical OR between these values. In other words, the query will search for records that contain any of the search values.

For example, you might enter search values such as these:

```
COMPANY  ACME
          HAUSER
```

This creates a search for all records in which the company name is either ACME or HAUSER.

On the other hand, if you enter search values into more than one field, the query will assume a logical AND between fields. This means that you will be requesting records that meet all search criteria.

For example, you might create a search using these values:

```
COMPANY  ACME
ZIP      >=50000
```

In this case, the query will be constructed to search for those records in which the company name is ACME and the ZIP code is greater than or equal to 50000. Any records to be found must meet both criteria.

Reserved Characters

Occasionally, you might wish to base a search on data that includes the special characters listed in Table G3-1 earlier in this section. For example, you might wish to find a company whose name includes the % (percent character).

If you look at Table G3-1, you will find that % (percent character) is a reserved character meaning "search for a specific pattern". This causes a problem, since the Browse Cross Reference window will interpret your percent character incorrectly.

In order to overcome this problem, you can repeat a reserved character to indicate that it is a part of the search data. If the reserved characters used in the Browse Cross Reference window are repeated, they are treated as literal characters, not special characters.

For example, imagine that you have a customer whose company name is "The %-age Company". If you try to create a search using the syntax:

```
COMPANY  %-age
```

the Browse Cross Reference window will interpret your search as a matching search, and will attempt to resolve "-age" as a match criterion. However, you can use this syntax:

```
COMPANY  %%-age
```

Here, the two percent characters together indicate that the percent character is to be treated as a literal character, and not as a matching search.

Repeating characters in this fashion works for all the reserved characters except the ellipsis (...). This symbol cannot be made a part of the search data.



Chapter G 4

Using Query

This chapter explains how to use Advanced Revelation's Query facility to look up records from a data file in the same windows you use for data entry.

Using Query in Application Windows	G 4 . 1
Using Query from TCL	G 4 . 9
Modifying the Environment for Query	G 4 . 11



Using Query in Application Windows

With Advanced Revelation's Query facility, you can easily look up records from a data file in the same windows you use for data entry. With Query, any data entry window becomes a form that you simply fill out with the search and sort criteria you wish to use.

Query takes the information you enter into a data entry window and constructs an R/LIST SELECT command. When you press [F9], the SELECT command is executed. Any records found by query are returned as a browse list into the data entry window.

Query supports most R/LIST selection features. You can specify selection criteria, including all operators such as equal, less than, greater than, starting with, ending with, containing and matching. Query also supports multiple selection criteria linked with AND and OR, as well as multiple sort criteria.

Initiating Query (\)

To use Query in any data entry window, enter \ (backslash) then press [Enter] at the key prompt. This will turn the window into a form in which you can move the cursor from prompt to prompt, entering selection and sort criteria.

In Query, the title of the screen in the top border adds the words “– Query Window” after the original title of the window. This indicates that you are in Query.

Moving between Prompts

While in Query, you can use the cursor arrow keys ([↑], [↓], etc.), the [Enter] key, or the [Alt-A] prompt popup to move from prompt to prompt. These are the same keys that are active in a normal data entry window.

If your window can normally use prompt or page tabs, you can use these in Query as well. For example, if you can use [Alt-T] (tab) and [Alt-Y] (backtab) or [PgUp] and [PgDn] in your window, these keys will also be active in Query.

You can move into symbolic prompts as well as normal prompts, but you cannot search on prompts that are defined in the window as joined fields.

For more information on cursor movement in windows, see the “Entering and Editing Data in Windows” chapter in *Getting Started*. For information on joined fields, see “Joining Prompt Fields with Other Files” in the “Adding Advanced Features to Window with Paint” chapter in *Developing an Application*.

Creating the Query

To create a query, initiate Query by pressing \ (backslash) at the key prompt. Then move the cursor to the appropriate prompt(s) and enter the data to be used for the query.

For most prompts, Query will accept input at the prompt entry line. When the cursor moves to that prompt, Query will display a special character mask that is as long as the prompt entry line defined in the window.

Hidden prompts or prompts for which the label has been deleted will prompt for input in the bottom border of the window. In place of a prompt label, Query will use the dictionary name associated with that prompt.

If you need more space to enter selection and sort criteria, use the [F3] key to zoom the prompt into a separate window. This gives you more room to enter information. After you have entered selection and sort criteria, use [F9] to save your changes and exit the Zoom window. You will be returned to the Query window.

Press [Esc] at any prompt to quit Query and return to the original data entry window.

Executing the Query

Once all selection and sort criteria have been entered, press the [F9] key. Query will construct and execute an R/LIST SELECT command.

Any records found by the query will be returned as a browse list into the original data entry window. If no records are returned, Query returns to the original data entry window with no browse list. Any browse list you might have had active before you began your query will be cleared.

Entering Selection Criteria

For most searches, simply enter the data at the prompt. To specify an operator other than = (equal), enter the operators in front of the search data.

For substring searches (starting with, ending with, or containing), or for range searches (from..to), embed the operators for these searches directly into the search data.

The full set of operators available in Query is listed below.

Table G4-1

Operators Available in Query

Operation	Method	Operator	Example
Comparison	Equal	null or =	=50
	Greater than	>	>99
	Less than	<	<100
	Greater than or equal	>=	>=100
	Less than or equal	<=	<=1000
Substring	Beginning with	...]	ACM]
	Ending with	[...]	[ING
	Containing	[...]	[TON]
Range	From-To	...	25...50
Pattern Match	Matching	%	%3N'-4N
Logical	Not	#	#>1000 (not greater than 1000)

For example, to enter criteria at a prompt called *Company*, you might enter one of the following:

Data	Meaning
ACME	Company equal to ACME
ACM]	Company starting with ACM
>=ACME	Company greater than or equal to ACME
%5A	Company matching 5 alpha characters
#[INC]	Company not containing INC
ACME...DIAMOND	Company from ACME to DIAMOND

For more information on the use of operators (comparison words) in selection criteria, see **WITH** in the “R/LIST Keyword Reference” chapter in the *R/LIST* section.

You should enter information at the prompt exactly as it appears in the database. For instance, if you have stored names in your database using upper and lowercase (for example, Smith), your search should be constructed the same way. Conversely, if data in the file is all uppercase (for example, SMITH), the information in the Query window should be entered in uppercase.

The data need not be in quotes; if quotes are embedded in a search value, they will be included as a part of the search. Beginning and ending spaces are trimmed from the data, but embedded spaces are considered a part of the search data.

Multiple Criteria for a Single Prompt

You can enter more than one selection value for a single prompt by separating each value with ; (semicolon) or & (ampersand).

Separating search values with ; (semicolon) will cause the values to be linked with a logical OR in the search. Separating the values at a prompt with & (ampersand) will cause those values to be combined with a logical AND.

For example, you might construct searches like the following:

```
Company      ACME ; MILLER
Contact      [SMITH] & [JACK]
```

The first example will search the file for all records in which the company name is ACME or MILLER; either would be acceptable.

The second example will search the file for those records in which the contact name contains both the name SMITH and the name JACK. In other words, the search will find records in which the contact name is JACK SMITH.

The second example also illustrates that you can use operators (for example, [], <, etc.) when linking data. Simply include the operator by placing it directly in front of, or embedded in, the data. For example:

```
ZIP          <10000 ; 80000...90000
```


Using AND and OR between Prompts

If you enter selection criteria for more than one prompt, Query links these prompts with AND. For example:

```
Company    ACME
ZIP        50000
```

Query creates a search in which the company name is ACME and in which the ZIP code is 50000. Both criteria must be met for a record to be selected.

If you wish to link selection criteria for different prompts with an OR, use a semi-colon (;) as the first character at that prompt. For example:

```
Company    ACME
ZIP        ;>50000
```

This query will find records in which the company name is ACME or the ZIP code is greater than 50000. If either criterion is met, the record will be selected.

Entering Sort Criteria

In addition to creating selection criteria, Query can include specifications for sorting. To indicate that a prompt is to be used in sorting, use the keyword BY. The keyword BY can be included anywhere in the data entered for a prompt. For example:

```
ZIP        BY
```

Entering the word BY in this way will cause Query to create a search in which the records will be sorted according to ZIP code.

If you wish to sort by multiple criteria, all BY keywords should be followed immediately by a number for the sort priority. Thus, to indicate that a prompt is to be the primary sort prompt, the phrase should read BY1. Indicate additional sort levels by including the phrases BY2, BY3, etc., at different prompts. For example:

```
ZIP        BY1
Status    BY2
```

In this query, the records will be sorted first by ZIP code, and then within like ZIP code values, by STATUS.

To create a sort in descending (reverse) order, use the keyword BY-DSND instead of BY. For example:

```
Order Date    BY-DSND
```

This query produces a report in which the records appear in descending order – in other words, with the most recent order date first.

If more than one descending sort is required, you can indicate a sort priority. The convention is the same as for BY. For instance, to indicate that the second-level sort is to be descending, enter the keyword BY-DSND2 at a prompt.

Editing the SELECT Command (Shift-F1)

Press [Shift-F1] while in Query to display the Query Select window. This window contains the R/LIST command that Query has constructed using the search and sort criteria you entered.

You can use the Query Select window to edit the command. For example, you might wish to change the sort order of prompts. Rather than entering and changing the appropriate prompts in the Query window, you could do this in the Query Select window. You might also wish to change the parentheses that Query builds in the R/LIST command.

If you are an R/LIST novice, you can use Query as an R/LIST tutorial. Construct searches using Query, and call the Query Select window often to display the correct syntax of the resulting R/LIST command.

To save any changes you have made in the Query Select window, as well as to execute the Query, press [F9].

- ☛ If you change the R/LIST command while in the Query Select window, you cannot return to Query in the data entry window.

When you save the command, you will be given an opportunity to save your query as a filter. See “Saving the Query as a Filter” below for details about this function.

If you press [Esc] at the Query Select window, you will be returned to the Query entry window, and can continue to enter selection and sort criteria. If you have made changes while in the Query Select window, these changes will not be saved.

Saving the Query as a Filter (Shift-F2)

At any point while you are creating a search in Query you can save the current query as a filter. To do so, press [Shift-F2]. A window will appear prompting you for the name of the filter under which you wish to save the query.

Enter a filter name without any spaces. The R/LIST SELECT command that Query has created out of your search criteria will be saved in the FILTERS file using the name you specify.

To access the filter later, use the standard Advanced Revelation filter access methods. For more about using filters, see the “Using the Filter Key” chapter.

Redisplaying the Last Query

Query remembers the last query you constructed and executed. You can recall this last query if it was created in the same window and if you have not logged off or changed accounts since you created it.

To recall the entire last query (except for changes you made in the Query Select window), enter Query by typing in `\\` (two backslashes) and [Enter] at the key prompt. Query will display all the selection and sort criteria you last used.

You can also recall the last query by entering Query normally using `\` (single backslash). Once you are in Query, press [Alt-C] to recall the entire last query.

To recall the selection or sort criteria for a single prompt, press [Alt-O] at that prompt. Query will fill in the criteria you last used for that prompt.

Changes to Prompts in Query

Because you are not truly entering records when in Query, some prompt specifications and window processes for the window are not active. The following changes are made to prompts and other processes in a window while in Query:

- Multivalued and text prompts are treated as single-valued, one-line prompts.
- Output patterns and output masks are ignored.
- Pattern matches are disabled.
- Maximum length specifications are ignored.
- All prompts accept lowercase input.
- Individual detail help messages or processes are replaced with global Query help.
- Window-specific softkeys are replaced with softkeys for the Query process.
- The Relations key is disabled.
- Pre-prompt and post-prompt processes are disabled.
- Pre-processes, post-processes, and replace-processes are disabled.

Accessing Indexes in Query

If you have created indexes for your data file, you might be able to take advantage of them in your query. If so, the query will run faster.

Accessing Btree Indexes

Query (and R/LIST) will automatically make use of Btree indexes whenever practical. When you are searching or sorting a field, Advanced Revelation will determine if the field is indexed using a Btree index, and if so, will determine if it is efficient to use the index. You do not need to make any special provisions or add any commands in order for Advanced Revelation to use a Btree index.

For example, if you create a search in Query that looks for a particular range of ZIP codes, and if the ZIP code field has been indexed using a Btree index, the query will determine if it is efficient to use the index to find the appropriate records. If so, the query will simply find the records in the index. If ZIP code is not indexed in a Btree index, the query will be forced to look at every record in the file, examining each one in turn to determine if it fits the search criteria.

Accessing Cross Reference Indexes

In contrast to Btree indexes, Cross Reference indexes are not automatically used when you create a query. However, if you know that a particular field has been indexed using Cross Reference indexing, you can specify that the query use the index.

To specify that you wish the query to use a Cross Reference index, put a \ (backslash) in front of the search data.

For example, entering this at the company name prompt:

Company \ACME

will cause the query to look for the word ACME in the Cross Reference index. Under most circumstances, this will be considerably faster than looking through each record in the data file.

If you use a \ (backslash) in front of search data, Query will check whether the field has been indexed using Cross Reference indexing. If it has not, Query will simply ignore your request to use the index. Since this is the case, you can use a backslash anytime you suspect that you might be able to take advantage of Cross Reference indexing, even if you are not sure.

If you want to search for multiple values in a field, and you want to use a Cross Reference index for each of the values, each value in your entry must be preceded with \ (backslash). For example, if the field COMPANY is indexed using a Cross Reference index, and you want to search for the company names ACME and ABC, you would enter \ before each of these values:

Company \ACME; \ABC

If you want to include an operator (<, >, >=, etc.) when doing a search in a Cross Reference index, the operator should follow the \ (backslash). For example, this entry at the Company name prompt (COMPANY is assumed to be indexed using a Cross Reference index) will create a search for records in which each word in the company name field (less those in the stop list) starts with a letter higher than M:

Company \>=M

- ☞ To determine whether data has been indexed in a Cross Reference index, use the **Tools-Files-Indexing-Listindex** options from the Advanced Revelation Main menu, or the LISTINDEX command at TCL. For information on LISTINDEX, see the “TCL Command Reference” chapter in the *TCL* section.

Cross Reference Indexes and Stop Lists

When you use a Cross Reference index in a query, you should be aware of the stop list used in that index. A stop list is a list of words that are considered too common to index when the Cross Reference index is created. A stop list typically includes words such as “THE,” “INC,” “AND,” etc.

If you inadvertently use a Cross Reference index for a query involving a word on a stop list, the query will not find the word, since it will not be in the index.

For more information on stop lists, see “Cross Reference Indexing” in the “Creating and Maintaining Indexes” chapter in *Developing an Application*.

Using Query from TCL

You can call a window directly from TCL and display it with Query active. To do so, enter the command QUERY followed by the name of the window. For example, the command:

```
QUERY CUSTOMER_ENTRY
```

will cause the CUSTOMER_ENTRY window to be displayed in Query. You will not need to enter a \ (backslash) in order to put the window into Query.

Enter selection and sort criteria normally. When you have finished constructing your query, press [F9] to save the query and execute it. If you press [Esc] before executing the query, you will be returned to TCL.

When the query is finished, you will be returned to TCL. The select list generated by the query will be returned to TCL as an active select list, rather than being returned as a browse list into the data entry window. The active select list can then be used in a subsequent process, such as a LIST command, an R/BASIC program, etc.

If your query produced no results, there will be no select list active when you return to TCL.

Examples of Query Syntax

Prompt	Data	Meaning
<i>State</i>	#WA	State not equal to WA
<i>Company</i>	[COMPUTER]	Company containing COMPUTER
<i>ZIP</i>	#>70000	ZIP not greater than 70000
<i>Phone</i>	%3N'-' 3N'-' 4N	Phone matching the pattern: 3 numbers, dash, 3 numbers, dash, 4 numbers
<i>Invoice Date</i>	1/1/87... 12/31/87 BY	Invoice date from 1/1/87 to 12/31/87 by invoice date
<i>Contact</i>	\JAMES	Contact equals JAMES (force Cross Reference index lookup, if the index exists)
<i>Status</i> <i>Last Order Date</i>	INACTIVE ;<1/1/86	Status = INACTIVE or last order date less than 1/1/86
<i>City</i> <i>State</i> <i>Area Code</i>	A] ; B] BY2 BY1 BY3 <300	City starting with A or City starting with B and Area code less than 300 by State by City by Area Code
<i>President</i>	[TRUMAN] ; [QUINCY] & [ADAMS]	President containing TRUMAN or President containing QUINCY and ADAMS

Table G4-2

Quick Reference Chart of Query Syntax

Operation Type	Operator	Operation
Entering and Exiting	\	Enter Query
Query	\\ [Alt-C] [F9] [Esc]	Recall last query Recall last query Execute query Exit without saving
Moving between prompts	[Alt-A] [↓][↑] [Enter]	Moved to selected prompt Move to next/previous prompt Move to next prompt
Editing in Prompts	[F3] [Alt-O]	Expand prompt edit area (zoom) Copy prompt value from previous query

(continued)

Table G 4-2
Quick Reference Chart of Query Syntax (Continued)

Operation Type	Operation	Operation
Selection Operators	= or null	Equal
	>	Greater than
	<	Less than
	=	Greater than or equal to
	<=	Less than or equal to
	#	Not (not equal to)
	[...]	Starting with
	[...]	Ending with
	[...]	Containing
	...to	From..to
	%	Matching
	;	OR
	&	AND
Sort Criteria	\ data	Use Cross Reference index for searching, if applicable
	BY	Sort BY this field
	BYn	Sort BY this field, priority "n" (example, BY2 = secondary sort field)
	BY-DSND	Sort BY this field in descending order
Softkeys	[Shift-F1]	Display Query Select window (edit search command)
	[Shift-F2]	Save Query as filter

Modifying the Environment for Query

Three prompts in the Query Window Environment window enable you to customize aspects of Query's appearance and function. To access the Query Window Environment window, choose **Management-Environment-Query** from the Advanced Revelation Main menu.

Customize Selection Editor

Use a code and command at this prompt to substitute your own window or process for the Query Select window called with [Shift-F1] from Query. For example, you might substitute your own collector window (with a "W" code) or your own sub-routine (with an "S" code) for the default Query Select window.

The query constructed in the window will be passed to your process in @PSEUDO. The word SELECT, file name and selection criteria will be in @PSEUDO<1>, and the sort criteria in @PSEUDO<2>.

For example, if you had entered search criteria to construct the command:

```
SELECT CUSTOMER WITH COMPANY = "ACME"
BY ZIP BY CITY
```

@PSEUDO would be passed to your routine with these values:

Variable	Value
@PSEUDO<1>	SELECT CUSTOMER (WITH COMPANY = "ACME")
@PSEUDO<2>	BY ZIP BY CITY

Once your routine is finished, Query will reconstruct the SELECT command based on the contents of @PSEUDO. If you set @PSEUDO to null, no SELECT command will be executed.

Border Prompting

If this prompt reads N, Query will prompt for selection and sort criteria at the window's prompt entry line (except for prompts with hidden labels). If the prompt reads Y, Query will use the bottom border to prompt for criteria. You might find border prompting more convenient if many prompts in a window have short display lengths.

Masking Character

You can specify the mask character to use to show prompt entry lines when Query is active. Indicate the ASCII value of the character you wish to use (32 for space). The default is ASCII character 176.

Chapter H 1

Creating Reports with Advanced Revelation

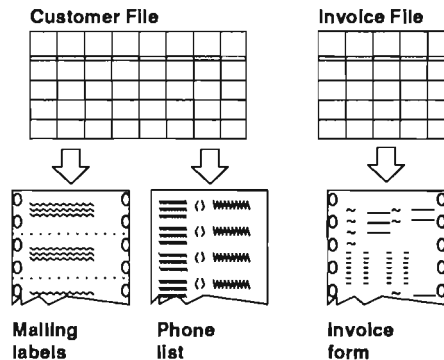
This chapter introduces the Advanced Revelation tools used to produce reports. It also explains how to customize the data dictionary for reporting purposes and how to use the View window to examine reports.

Overview of Advanced Revelation Reports	H 1 . 1
Overview of Advanced Revelation Report Tools	H 1 . 2
Customizing the Data Dictionary for Reports	H 1 . 2
Using View for R/LIST Reports	H 1 . 4
Developer Notes for View	H 1 . 6



Overview of Advanced Revelation Reports

Advanced Revelation offers you four kinds of reports: forms, labels, lists, and merge reports (reports that combine and format text with varying quantities of data). Each type of report can be predefined for use in an application. For example, a sales application might require reports that produce invoice forms, customer mailing labels, a phone list, and sales letters personalized for each customer. Each of these reports would print information from a particular file. Invoices would be printed using data from the Invoices file, while the other reports might draw their data from the Customer file.



- **Forms** are often used when you want to arrange the layout of fields in your report. For example, you can lay out your fields to match paper forms, such as insurance forms, and print data to fill in the blanks. Successive forms on a page print in rows across or columns down the page. You can print one or more records per page.
- **Labels** are most often used to create mailing labels. You specify the number of lines for the label and how many labels to print across a page. Labels are also convenient for producing address lists for reference or for editing.
- **Lists** are often used when you want to see the contents of a file in columns and rows. Lists can also show relationships between records by displaying totals or by grouping records into categories.
- **Merge Reports** are best suited to creating forms for variable data. You can merge data from an Advanced Revelation file into a form or document that also includes specified text, variable information such as date and time, headings and footings, and character fonts such as boldface and italic.

Overview of Advanced Revelation Report Tools

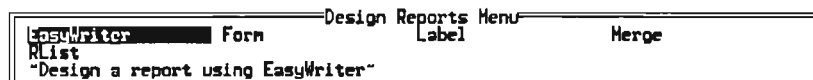
The tools used to design a report format are listed as options on the Design Reports menu.

When you design a report, you create a template for the report and save it in a file.

- Forms templates are saved in the REPORTS file.
- Labels templates are saved in the LABELS file.
- Lists templates are saved in the QUERIES file.
- Merge templates are saved in the MERGES file.
- After creating the format, you can print the report through the Access menu or from The Command Level (TCL).

The Reports Menu

To create a report format, choose **Design** from the Main menu and **Reports** from the Design menu. The Design Reports menu appears:



- To create a Form report, choose **Form**.
- To create labels, choose **Label**.
- To create a List report, choose **RList** or **EasyWriter**.
- To create a Merge report, choose **Merge**.

To find out how to use each of the report design tools, see the appropriate chapter within *Creating Reports*.

In addition to the tools used to create reports, Advanced Revelation provides a special View window that can be used to scan R/LIST reports at the terminal. For more information, see “Using View for R/LIST Reports” later in this chapter.

Customizing the Data Dictionary for Reports

The data dictionary tells Advanced Revelation what to do with the data that is stored in a field. When Advanced Revelation prints an R/LIST or Label report, it uses information in the data dictionary to determine the display width and justification of each field. You can adjust these settings using the Dictionary window. In addition, if you want custom column headings printed in an R/LIST report, you can create the headings in the Dictionary window.

This section explains how to use prompts in the Dictionary window to specify the field width and justification that will display in R/LIST and Label reports and to customize column headings in the report.

Displaying the Data Dictionary

To display a field's data dictionary from the Main menu choose **Tools-Files-Dictionary**. The Dictionary window appears.

Setting Output Format

Use the *Output format* prompt to control the data's output pattern. You can tell Advanced Revelation to display the data in a specific format, regardless of how the data was entered. For example, you can specify that Advanced Revelation is to display a date in the format mm-dd-yy. Regardless of what format was used to enter the date, Advanced Revelation displays the date in the pattern you set.

For more information on output patterns, see "Data Formatting" in the "Adding Advanced Features to Windows with Paint" chapter in *Developing an Application*.

Adjusting Field Justification and Length

Use the *Justification* and *Display length* prompts to control the field's justification and the length of the output lines.

Justification By default, Advanced Revelation displays data in a field with left justification. To change the justification of a field, enter the appropriate option at the *Justification* prompt. To see all the options, press [F2].

For example, if all of the values in a field are a single digit, you may want the values centered under the field heading. In that case, enter C at the prompt.

Display Length By default, Advanced Revelation allows ten spaces for the length of a field when the field is printed. To change the length, enter the appropriate number at the *Display length* prompt.

Customizing a Column Heading

By default, Advanced Revelation uses the field name as a column heading on R/LIST reports. To change the heading, type the heading you want at the *Column heading* prompt.

Using View for R/LIST Reports

Advanced Revelation's View option displays R/LIST reports in a special View window. Reports in this window can be paged forward and backward, and reports wider than the screen can be panned side-to-side. This permits you to scroll through an entire report in both directions. In addition, you can display reports wider than the screen without causing lines to wrap to the next line.

Displaying a Report Using View

When View is active, any R/LIST report you create will be displayed in a special View window. R/LIST reports can be generated from TCL using LIST or SORT, from the Rlist window, from a special R/LIST processor such as EasyWriter, or directly from a window using [Alt-V]. For more information on creating reports using R/LIST, see the "Creating R/LIST Reports" chapter and the "Writing Reports with EasyWriter" chapter in *Creating Reports*. Also see "Introduction to R/LIST" in the *R/LIST* section.

If the report is narrower than the screen, the View window will be correspondingly smaller. If the report is wider than the screen, the window will show a dashed border, with the hidden portion of the report off the dashed edge of the screen. This hidden portion can be seen by panning in the appropriate direction.

View has no effect on reports directed to a printer.

- ☛ View is activated by a setting in the General System Configuration window. The default is that View is active. Details on this setting and on other settings for View can be found in the "Developer Notes for View" section in this chapter.

Keys Used in View

When a report is displayed in a View window, special keys enable you to pan forward and backward and side-to-side in your report. Other keystrokes change the number of columns shifted when you pan, permit you to jump to a specific page of the report, enable you to modify your report, or route your report to the printer.

Summary of Keystrokes Active in View

Type	Key	Meaning
Movement Keys	[Enter], [↓], [PgDn]	Display next page of View Report
	[↑], [PgUp]	Display previous page of View report
	[Ctrl-PgUp]	Display first page of report
	[Ctrl-PgDn]	Display last page printed so far
	[Home]	Pan report to first column
	[End]	Pan report to last column
	[←]	Pan to show additional columns left
	[Ctrl-←]	Pan left one window width
	[→]	Pan to show additional columns right
	[Ctrl-→]	Pan right one window width
Special Keys	[Spacebar]	Increment number of columns shifted while panning left-right
	[Backspace]	Decrement number of columns shifted while panning left-right
	[Ctrl-G]	Go to a specific page of the report printed so far
	[Alt-V]	Modify and restart report
	[Alt-P]	Print the current report on the printer from the beginning

Modifying and Restarting the View Report (Alt-V)

To modify and re-execute the View display while in a View window, press [Alt-V]. View will display the current R/LIST sentence in an R/LIST window.

You can change the R/LIST sentence displayed in the R/LIST window by editing it. When you press [F9] at the window, the current report will be regenerated from the beginning of the report with the changes you have made in the R/LIST window.

- ☛ If your original report was running from a list of record keys (a select list), the select list will not apply to your modified report. For example, if you had used GETLIST to activate a select list before running your original report, that select list will no longer be active for the modified report.

Routing a View Report to the Printer

If you press [Alt-P] while in a View window, the current report will be routed to the printer. The report will be regenerated, and then printed from beginning to end without pausing.

While the report is printing, you cannot pan the report, nor can you use [Alt-V] to modify the display of the report. However, once the printed report is finished, you will be returned to the place in your View window from which you pressed [Alt-P], and can continue viewing the report.

Developer Notes for View

This section provides background information about the behavior of View, and about customizing View. Developers can disable the View window, and can customize the use of the [Alt-V] key. In addition, the screen and printer sizes assumed by View can be changed.

The View Operating System File

View maintains an operating system file for screen images of pages that have already been displayed. As more pages are displayed, the file grows. The operating system file is deleted when the report is done.

The View operating system file is maintained in the root directory of the current default drive. The operating system filename is constructed according to this formula:

```
"ROS" + station.id (5 characters) + ".VF" + TCL Level (1 character)
```

For example, a View report from TCL level 2 for station 12345 will create a View operating system file called ROS12345.VF2.

Warning! If you alter the contents of this file while your report is being displayed, unpredictable results will occur.

The size of the operating system file required by View is dependent on the width and height of the report. The formula for calculating the eventual size of the operating system file required for View is:

```
(number of columns * 2) * (number of rows) * (number of pages)
```

For example, a report that is 80 columns wide and that prints 20 rows on a screen requires 3,200 characters per page. A report of 10 pages will require an operating system file of 32,000 bytes.

Changing the View Operating System Path

Use the **Management-Environment-General** options from the Advanced Revelation Main menu to access the General System Configuration window. View puts the View operating system file on the drive or volume pointed to by the *Sort DOS path name* prompt in the General System Configuration window. If this prompt is blank, View will put the operating system file on the root directory of the current drive.

Changing this volume name to that of a fast hard disk or RAM disk will cause View to run faster.

Deactivating View

View is activated or deactivated by a setting in the General System Configuration window. Use the **Management-Environment-General** options from the Advanced Revelation Main menu to access the General System Configuration window. Entering N at the prompt *View Mode On* will cause R/LIST reports to display without a View window. In this mode, R/LIST reports will not be saved to a View operating system file, and therefore will not permit paging backward. In addition, reports wider than the screen width will wrap around to the next line.

Deactivating View might be desirable in situations where disk space is limited and there is no room for a View operating system file.

Setting Window and Screen Width for View

View calculates the width and height of the View window based on the current settings for @CRTWIDE and @CRTHIGH. These settings can be changed using the SET-CRT command from TCL, by changing the settings in the Screen and Printer Display window, or by modifying these variables directly from within an R/BASIC program. To access the Screen and Printer Display window, use the **Management-Environment-Display** options from the Advanced Revelation Main menu.

Specifying a Custom R/LIST Window or Process

Use the **Management-Environment-General** options from the Advanced Revelation Main menu to access the General System Configuration window. By entering a code and command at the *View Mode Sentence Builder* prompt in the General System Configuration window, you can reprogram the [Alt-V] key to do something other than display the default R/LIST window. Your options include calling an alternate window (using a "W" code and command), or a custom routine (using an "S" code and command, for example).

If you create a custom process or window that will generate an R/LIST sentence, put the finished sentence into the variable @PSEUDO. When your process terminates, View will run whatever it finds in @PSEUDO as an R/LIST sentence. If @PSEUDO remains null, View will simply return to the original View window.

- ☛ The code and command filled in at the *View Mode Sentence Builder* prompt in the General System Configuration window is also used when a user presses [Alt-V] in an application window. If you change this code and command, be sure that this will not affect window processing for your applications.



Chapter H 2

Creating Forms

This chapter explains how to create form reports using Advanced Revelation's Paint tool. From Paint you can define the data file used by the form, the form characteristics, and even test run a form to check that everything is working as it should.

Creating Forms	H 2 . 1
Specifying Form Characteristics	H 2 . 7
Testing a Form	H 2 . 10
Printing Forms	H 2 . 10



Creating Forms

Advanced Revelation's Paint tool enables you to create forms to match your data to an existing paper form or to create a layout to your own specifications. While using Paint to create a form, you can also define the file that will hold the form's data. This chapter describes how to build forms using Paint. For more information on Paint, see "Creating Windows with Paint" and "Adding Advanced Features to Windows with Paint" in *Developing an Application*.

Form reports are commonly used when you want to place fields in a particular layout. You use Paint to create forms in much the same way as you use it to create windows. This section describes the features in Paint as they relate specifically to form definition.

While defining your form, you determine the display width of each field, the number of records printed across the form, and how many spaces Advanced Revelation leaves between records and rows. In this way, you determine the size of the printed form.

Using Forms to Design File and Record Structure

If you are creating a new application or file, you can use a form design to help you design the file that holds its data. In fact, you can design all your reports first, and then design the application based on its reports.

This approach is particularly convenient when building form reports, because you can create your file at the same time as you create your form.

First, determine what information will appear on the form. Decide where and how you want the information displayed. Next, create your form and file at the same time in Paint.

Naming the Form and File

To create or modify a form, choose **Design-Report-Form** from the Advanced Revelation Main menu. Advanced Revelation displays the Paint Form window.

Modifying an Existing Form

If you want to modify an existing form, enter the form's name. To choose from a list of available forms, press [F2] (Options).

When you enter the name of an existing form, Paint displays the form definition so that you can change it.

Creating a Form

If you want to create a new form, enter the new form name. Your form name can contain up to 50 alpha-numeric characters, including dollar signs (\$) and underscores (_).

Paint centers the form name at the top of the screen. Then Paint displays another window, to prompt for a file name.

You can create a form that reads an existing data file or you can use Paint to design a new data file for your form to use.

Using an Existing Data File

To create a form that will access an existing data file, enter the name of the file. You can press [F2] (Options) for a list of available files.

Advanced Revelation then displays your form in Paint.

Creating a New File

To create a new file, enter the new file name. The filename may contain up to 50 alphanumeric characters, including dashes, dollar signs and underscores. Do not use periods or parentheses in the filename and do not begin the file name with a number. The filename ends at the first occurrence of a space.

For compatibility with SQL, we recommend that you follow these conventions when specifying a filename:

- use up to 18 characters
- begin with an uppercase alphabetic character
- use no punctuation except underscore (_)
- use only alpha and numeric characters (plus underscore)

After you fill in this window, press [F9] (Save) to save your entry.

If you are creating a new file, Paint displays the Make Dictionary window.

Your entries at the Make Dictionary window tell Advanced Revelation where to put both the data and dictionary portions of the new file when they are created. Help is available at any prompt in the Make Dictionary window. Just press [F1] (Context help) or [Ctrl-F2] (Concept help).

The new dictionary name appears at the *Dictionary name* prompt. To indicate where the dictionary and file should be located and the file type you want to create, enter a volume name at the *Volume* prompt. For a list of attached volumes, press [F2] (Options). If you do not make an entry at the *Volume* prompt, the dictionary and file will be created on the default data volume DATAVOL as an Advanced Revelation file.

- ☛ Your environment may specify a volume other than DATAVOL as the default data volume. The default data volume is specified at the *Default Data Volume* prompt in the General System Configuration window. To access the General System Configuration window, choose **Management-Environment-General** from the Advanced Revelation Main menu.

Depending on the file type that you are creating, you may want to specify attributes that apply to this file. To do so, press [F2] at the *File Options* prompt. Paint will display a collector window that prompts for file attributes. Press [F1] (Help) at any prompt in the window for help making a specification. Fill in the prompts and press [F9] to save your specifications.

- ☛ For information about the file attributes used with Advanced Revelation files, see the “Managing Files and Records” chapter in *Filing Systems*. If you are creating a bonded file, see your bond documentation for an explanation of the file attributes you can specify for that file type.

After filling in the Make Dictionary window, press [F9] (Save) to save your entry and create the dictionary portion of the file. Advanced Revelation then displays your form in Paint.

Using Window Templates as Forms

You can frequently save time and effort in defining a form by starting with the template of an existing window and modifying it to suit your reporting needs.

To use a window template for your form, first copy the window template record from the TEMPLATES file to the REPORTS file. Once the record is copied, use it “as is” to produce a form exactly like your window, or modify it as you wish.

Follow these steps to copy the definition:

1. From the Main menu, choose **Tools, Records, and RecordCopy**. Advanced Revelation displays the Recordcopy window.
2. Enter TEMPLATES at the *Source file* prompt and REPORTS at the *Destination file* prompt.
3. Enter the name of the window you want to copy at the *Source record(s)* prompt.
4. To use the window name for your form name, press [Enter] at the *Destination record(s)* prompt. To give your form a different name, enter the name you want to use.
5. Press [F9] (Save) to perform the command. Advanced Revelation then writes the window template record to the REPORTS file. Now you can return to Paint through the Design Reports menu and modify the form definition as you wish.

Using Virtual Space to Plan Your Page Size

You are not limited to the size of the screen in creating your report. In fact, the maximum area you can use contains 32,000 characters. If you use the maximum area in a square, that translates into a layout 180 columns wide and 180 rows deep. The size of your printed form is limited only by the size of paper your printer can handle, and the amount of memory you have available.

Using Paint Options to Create Forms

Use Paint options to create forms in the same way you use them to create windows. For details on Paint options, see the “Creating Windows with Paint” chapter in *Developing an Application*.

The Form Menu	To display the Form menu, press [F10]. The purpose of each menu option is described below:
Quick paint	Use this option to create form prompts using existing dictionary fields. For more information, see the “Adding Advanced Features to Windows with Paint” chapter in <i>Developing an Application</i> .
Test run	Use this option to test your current form design. For more information, see “Testing a Form” in this chapter.
Delete	Use this option to delete the current form. See “Deleting a Form” below.
Form Parameters	Use this option to review and modify formatting characteristics for the current form. For more information, see “Displaying Form Characteristics” in this chapter.
Overstrike	Use this option if you previously used the Hide label option to suppress any of the prompt labels on your form. OverStrike shows you the field name associated with each prompt display line on the form. The field names will show within the display lines. For more information, see “Showing Field Names in Display lines” in this chapter.

Saving a Form

To save your form definition, press [F9]. Advanced Revelation writes the form definition to the REPORTS file. Your form must contain at least one prompt and display line to be written to the REPORTS file.

Deleting a Form

To delete the current form's definition from the REPORTS file, press [F10] to display the Form menu and choose **Delete**.

Paint displays a message informing you that your current form is about to be deleted from the REPORTS file.

To delete the form, press Y. To keep the form, press any other key.

Leaving Paint

To save any changes that you made to the form template, press [F9] before leaving Paint. To leave Paint, press [Esc]. Paint asks you to confirm that you want to exit. Press [Esc] to confirm or press any other key to remain in Paint.

If you are creating a form, and the data portion of the file does not exist yet, Paint will ask if you want to create the data file at this time. If you are not ready to create the data file at this time, enter N. To create the data file, enter Y.

- ☛ It is a good idea to refrain from creating the data file until you are completely done designing the form. Once the data file is created, a bonded environment may not allow you to alter the structure of the file without deleting it first. For more information, see your bond documentation.

Hiding Prompts

You can suppress the display of a prompt on your form while retaining the prompt's display lines. Use this feature when you want to display the data without any accompanying text or with a label heading.

To hide a prompt, follow these steps:

1. Move the cursor inside the prompt and press [F4] to turn on the Editor.
2. Press [F2] to display the Prompt Options popup.
3. Choose **Hide label** from the Prompt Options popup. Paint hides the prompt.

Showing Field Names in Display lines

After hiding prompts on your form, you may want to view the field names in their display lines. To display the field names, press [F10] to display the Form menu and choose **Overstrike**.

When Paint displays your form, the field names appear in their display lines. You can continue defining your form with the field names displayed. If you edit a display line, the field name in the display line disappears. To display it again, repeat the above steps.

Changing Prompt Characteristics

You can determine how Advanced Revelation displays the data in the area defined by a prompt's display line by setting these prompt characteristics: *Display length*, *Justification*, and *Output format*.

Changing the Display Length

Display length refers to the size of the prompt's data display line. Advanced Revelation displays data on the form within the dimensions of the display line. Before you can change the size of a display line, the cursor must be in the display line and the Editor must be on. Use [F4] to toggle the Editor on and off.

Use the arrow keys to change the size of a display line. The right arrow key and the left arrow key expand and contract the right edge of the display line. The down arrow key adds more lines; the up arrow key deletes them.

Displaying the Prompt Window

The Prompt window is used to adjust a prompt's *Justification* and *Output format* characteristics. Use either of these methods to display the Prompt window:

- Move the cursor inside the prompt or its display line and press [Shift-F6].
- With the cursor inside the prompt or its display line and the Editor on, press [F2] to display the Prompt Options popup. Choose **Prompt window** from the popup.

Press [F1] at any prompt in the Prompt window for more information. Press [F2] for options to enter at the prompt. When your entry is complete, press [F9] to save it.

Adjusting Prompt Justification

The *Justification* prompt in the Prompt window controls the justification for the prompt's data. These are the format options:

Option	Description
L	Left justified. The data starts at the left border of the display line. Data that extends beyond the right border is not displayed.
R	Right justified. The data ends at the right border of the display line. Data that extends beyond the left border is not displayed.

Option	Description
T	Continuous text. Text will be formatted exactly as it was entered.
TN	Text nonjustified. Any special text formatting will not be retained.
C	Centered. The data is centered between the borders of the display line.

Advanced Revelation's default setting displays your data left justified. To change the format, enter the option you want.

Changing a Prompt's Output Format

Use the *Output format* prompt in the Prompt window to further refine how the data for the prompt is displayed on the form reports.

The *Output format* prompt sets a specific format for the output of the data, regardless of how the data was entered. For example, you can set this pattern for the output of a date: mm-dd-yy. Even if the date was entered in a different format (for instance, dd month yyyy), the form report displays the date in the pattern you set.

For more information on Output formats, see the "Adding Advanced Features to Windows with Paint" chapter in *Developing an Application*.

Specifying Form Characteristics

Just as you use prompt characteristics to determine how data for an individual prompt will be displayed on your form, you use form characteristics to determine how the entire form will be printed or displayed on your computer screen.

To define your form's characteristics choose the **Form Parameter** option on the Form menu. Paint displays the Make Form Parameters window.

The specifications at this window help you organize the information on your form. When you are finished making entries at the window, press [F9] to save them.

Determining How Many Records to Print on a Page

By default, Advanced Revelation prints one record on each page of your report. To print more than one record across your page and down your page, you must specify the number.

Number of records across page	To print more than one record across a page, enter the desired number at the <i>Number of records across page</i> prompt. Advanced Revelation prints as many records as you determine, side-by-side across the page.
--------------------------------------	--

**Number of records
down page**

To print more than one record down a page, enter the desired number at the *Number of records down page* prompt. Advanced Revelation prints as many records as you determine.

Ordering Records by Columns or Rows

If your form displays more than one record across the page, you can tell Advanced Revelation whether to order the records by row or by column. Ordering by rows means that the records will be in sorted order across the page; ordering by columns means that the records on the report will be in order down the column.

By default, Advanced Revelation orders the records across rows.

Use the *Record order* prompt to change your record order. These are the options for record order:

- Order records across rows
- Order records down columns

Ordering records across the page saves printing time. Advanced Revelation can begin printing as soon as it has formatted one row of values. If your records are ordered down the page, Advanced Revelation must format the entire page before it can begin to print.

Formatting a Form

By telling Advanced Revelation how many spaces to put between records, where to place the top and left margins, and what to print as a heading and footing, you set up your report's format.

**Columns between
records**

If your form displays more than one record across a page, use the *Columns between records* prompt to specify the number of spaces between the columns of records. By default, Advanced Revelation puts two spaces between the columns.

**Rows between
records**

If your form displays more than one record down a page, use the *Rows between records* prompt to specify the number of blank lines between rows of records. By default, Advanced Revelation puts two blank lines between the rows.

Top margin

Use the *Top margin* prompt to set your form's top margin. By default, Advanced Revelation prints the first line of your form on the fifth line from the top of the page.

For example, suppose you are printing your form on letter-head stationary. In order to leave enough room for your company's logo, you want Advanced Revelation to start printing on the tenth line of each page. You would enter 10 at the *Top margin* prompt.

Left margin Use the *Left margin* prompt to set your form's left margin. By default, Advanced Revelation starts the left margin five spaces from the left edge of the page.

For example, suppose you want a two-inch left margin for writing notes and comments on your report. If your printer prints ten spaces per inch, you would enter 20 at the *Left margin* prompt.

Form heading Use the *Form heading* prompt to print a heading at the top of each page of your form. Enter the text of the heading exactly as you want it printed on your form.

You can include HEADING options with your heading text when you want Advanced Revelation to insert a value in your heading such as the current date or page number. For example, the heading definition for a CUSTOMER form might be:

```
CUSTOMER FILE AS OF 'D' PAGE 'P'
```

This heading definition causes Advanced Revelation to insert the date and page number on your form.

Form footing Use the *Form footing* prompt to print a footing at the bottom of each page of your form. Enter the text of the footing exactly as you want it printed on your form.

As with headings, you can include FOOTING options in your footing when you want Advanced Revelation to insert a value in your footing. For example, this footing definition

```
TIME OF PRINTING: 'T'
```

causes Advanced Revelation to insert the current time and date in the footing.

To use an option, enclose it in single quotes at the place in your text where you want Advanced Revelation to insert the value. For a complete description of each HEADING and FOOTING option, see the "R/LIST Keyword Reference" chapter in *R/LIST*.

Sending a Form to the Printer or the Screen

Use the *Option* prompt to direct your form to a terminal (computer screen) or a printer. By default, Advanced Revelation directs the form to a terminal. Valid *Option* entries are:

- P Advanced Revelation prints your form on the printer.
- T Advanced Revelation displays your form on the terminal.

Testing a Form

The Test Run feature allows you to see how your current form design looks. If the data file that the form accesses already exists, you can use Test Run to print a test report using data from the file.

To test your form, press [F10] to display the Form menu and choose **Test Run**. Advanced Revelation displays the Test Run Form window.

At the first prompt, enter the number of records you want to print in your test run. By default, Advanced Revelation prints 10 records.

At the second prompt, enter P if you want Advanced Revelation to print your test on the printer. By default, Advanced Revelation displays the report on the terminal (computer screen). If you send the report to the printer, be sure your printer is on and ready to print.

When your entry is complete, press [F9]. Advanced Revelation then displays the report. When the printing is complete, Advanced Revelation returns you to your form definition in Paint.

Printing Forms

To print your form, choose **Access** from the Main menu and **Report** from the Access menu. Advanced Revelation displays the Access Reports menu. Choose **Form** from the Access Reports menu. Advanced Revelation displays the Form window.

Using the Form Window

A description of the prompts on this window follows. After you fill in this window, press [F9] to print the specified report.

File name At the *File name* prompt, enter the name of the file that holds the form report definition. By default, Advanced Revelation stores form definitions in the REPORTS file. To accept the default, press [Enter] at the prompt. If your report definition is stored in another file, enter the name of that file. For a list of available files, press [F2].

Form name At the *Form name* prompt, enter the name of the form definition. For a list of available forms, press [F2].

Record(s) Enter the keys for the records you want included on the report. For a list of keys, press [F2]. To use a filter to select the records printed on your report, press the Filter key [Ctrl-F10]. For more information on creating and using filters, see the “Using the Filter Key” chapter in *Accessing Data*.

To print all records on the report enter an asterisk (*).

Chapter H 3

Creating Labels

This chapter explains how to create label reports using the Make Label window. From the Make Label window, you can define the label report format, specify the actual label layout, and test run the current label design.

Creating Labels	H 3 . 1
Testing a Label Template	H 3 . 3
Printing Labels	H 3 . 3



Creating Labels

Use Labels for mailing labels or similarly formatted reports. You create labels using the Make Label window.

To create or modify a Label report, choose **Label** from the Design Reports menu. Advanced Revelation displays the Make Label window.

Specifying a Label Name and File

The first two prompts in the Make Label window are used to specify the name of the Label report and the file that contains the data displayed by the report. A description of these prompts follows.

Label definition name	Enter the name of the label you want to create or modify. Your label name can contain up to 50 alphanumeric characters, including dollar signs (\$) and underscores (_). For a list of available labels, press [F2].
File name	Enter the name of the file that contains the data you want displayed on the labels. For a list of available files, press [F2].

Setting the Label Size

Advanced Revelation provides a series of prompts to enable you to set the label size.

Lines on label	Enter the number of lines on your labels. For example, if your labels are one inch deep and your printer prints six lines per inch, you would enter 6 at this prompt.
Columns on label	Enter the number of columns on your labels. For example, if your labels are 3.5 inches wide and your printer prints ten spaces per inch, you would enter 35 at the prompt.

Sending Your Labels to the Printer or the Screen

Use the *Option* prompt to direct your labels to the terminal (computer screen) or the printer. By default, Advanced Revelation directs the labels to the printer. Valid entries are:

Option	Meaning
P	Advanced Revelation prints your labels on the printer
T	Advanced Revelation displays your labels on the terminal

Setting Number of Labels to Print Across

Enter the number of labels you want to print across a page at the *Labels across page* prompt. By default, Advanced Revelation prints one label across a page.

Setting the Spacing Between Labels

You tell Advanced Revelation how much space to leave between labels by indicating the horizontal and vertical spaces between your labels.

Horizontal By default, Advanced Revelation leaves one blank line horizontally between labels. To change this setting, enter the correct number of lines at this prompt. For example, if your labels are separated by three blank lines, you would enter 3.

Vertical If you are printing more than one label across a page, by default Advanced Revelation leaves one blank column between the labels. To change this setting, enter the correct number at this prompt. For example, if your labels were separated by two columns (or spaces), you would enter 2.

Defining Your Label's Layout

Use the bottom half of the Make Label window to define your label's layout. Place the fields where you want data displayed and enter any text you want printed on each label.

When you first enter the *Label layout* area, the Editor is off. To move between lines in the area using the arrow keys and [Enter], you must turn the Editor on. Use [F4] to toggle the Editor on and off.

To display a field in the label, you can position the cursor where you want the field to appear and type the field name between braces ({ }) or you can choose the field name from a popup.

To choose a field name from a popup, follow these steps:

1. Position the cursor on the line where you want data from the field displayed.
2. Press the options key [F2].
Advanced Revelation writes the word "field" at the *Label layout* area and displays a popup listing all the fields in the file.
3. Choose the field you want displayed at your current cursor location.

When you press [Enter] or [F9] to save your choice, Advanced Revelation displays the field at the *Label layout* area in braces. To display more than one field on a line, position the cursor where you want the field to appear and press [F2] again.

You can enter text by typing it directly in your label definition. For example, the following label definition prints the three fields CITY, STATE, and ZIP on one line, with a comma between CITY and STATE and with three spaces between STATE and ZIP:

```
{CONTACT}
{COMPANY}
{ADDRESS}
{CITY}, {STATE}   {ZIP}
```

Testing a Label Template

To test your current label layout, press [Shift-F1] (Test Run). Advanced Revelation displays the Test Run Label window.

At the first prompt, enter the number of records you want to print in your test run. By default, Advanced Revelation prints 10 records.

At the second prompt, enter P if you want Advanced Revelation to print your test on the printer. By default, Advanced Revelation displays the report on the terminal (computer screen). If you send the report to the printer, be sure your printer is on and ready to print.

When your entry is complete, press [F9]. Advanced Revelation then displays the report. When the printing is complete, Advanced Revelation returns you to your label definition in the Make Label window.

Printing Labels

You have the option of printing your labels directly from the Make Label window. Press [Shift-F1] to initiate a test run. To print the labels, save the label definition by entering Y at the "OK to save?" message window. The Test Run Label window appears. Enter the number of sample labels to print and direct the labels to the terminal (T) or printer (P). Press [F9] to print the labels. If you are printing on the printer, be sure your printer is on and ready to print.

To print your labels at any other time, choose **Label** from the Access Reports menu. Advanced Revelation displays the Access Label window.

Using the Access Label Window

A description of the prompts on the Access Label window follows. After you fill in the window, press [F9] to save your entry and print the labels.

Label name Enter the name of the label definition. For a list of available labels, press [F2].

Options Use the *Options* prompt to direct your labels to the terminal (computer screen) or the printer. By default, Advanced Revelation directs the labels to the terminal. The valid entries are:

Option	Meaning
P	Advanced Revelation prints your labels on the printer
T	Advanced Revelation displays your labels on the terminal

To use a filter to select the records printed on your labels, press the Filter key [Ctrl-F10]. For more information on creating and using filters, see the “Using the Filter Key” chapter in *Accessing Data*.

Chapter H 4

Creating R/LIST Reports

This chapter explains how to create R/LIST reports from report templates designed at the Make an RList Report window. R/LIST templates enable you to produce columnar reports from any file currently available. An R/LIST report template can be used over and over. When you retrieve and re-execute the report template, the report it creates is generated from scratch, using the most current data.

Creating R/LIST Reports H 4 . 1

Using the Make an RList Report Window H 4 . 1



Creating R/LIST Reports

An R/LIST report displays the contents of a file in columns and rows. Each record in the file is printed as a row. The fields from the records are printed in columns. The report can show relationships between records by displaying totals or by grouping records into categories. For example, you might use an R/LIST report to create a telephone list that displays records from the CUSTOMER file and groups the records by area code.

A columnar report is created by executing an R/LIST LIST command. The LIST command can include options for selecting, sorting, and displaying records in the file.

You can define and store LIST commands as report templates. Once a report template is created, it can be used at any time to produce a formatted columnar report using the most current data. Report templates might be defined and included in applications where a standard group of reports must be produced routinely. For example, a sales application might include a report template for a weekly report that lists the sales generated by each salesman. A monthly follow-up report might list, by territory, customers who have not made a purchase in the last three months.

Developing Report Templates

R/LIST report templates are developed using the Make an RList window. The Make an RList window enables you to make use of all of the options available in an R/LIST LIST command.

The EasyWriter report generator also allows you to create a LIST command and store it for use as a report template. There are certain advanced features of R/LIST, however, that EasyWriter cannot use. For more information about EasyWriter, see the “Writing Reports with EasyWriter” chapter in *Creating Reports*.

Using the Make an RList Window

Use the Make an RList Report window to create, modify and delete templates used to produce R/LIST reports. R/LIST is Advanced Revelation’s English-like retrieval language. By entering simple R/LIST sentences, you will be able to display data from any file, and specify such attributes as:

- Which fields to display
- What order to display the records
- Which records to display
- Headings, footings, and column headings

By using this window, you will create an R/LIST template that can be used over and over.

Accessing the Make an RList Window

To access the Make an RList window, choose **RList** from the Design Reports menu.

Choosing a File Name and Report Name

The first two prompts in the RList window are used to specify the file that will hold the report definition and the name of the report.

RList template file name By default, Advanced Revelation stores the definitions of R/LIST reports in the **QUERIES** file. To accept the default, press [Enter] at this prompt. To store the definition in another file, enter the name of the file.

Template name Enter the name of the report you want to create or modify. Your report name can contain up to 50 alphanumeric characters, including dollar signs (\$) and underscores (_). For a list of available reports, press [F2].

Specifying a LIST Statement

Use the bottom half of the window to build the LIST statement that defines your R/LIST report. This is the general format of a LIST statement:

```
LIST {n} {ONLY} {DICT} filename {USING alt_filename}
  {selection_criteria} {sort_criteria} {fields}
  {keywords} {options}
```

A brief description of commonly-used parts of this statement follows. For a complete discussion of the elements in a LIST statement, see the *R/LIST* section.

When your LIST statement is complete, press [F9] to save it.

filename Names the file that holds the data you want to display.

selection_criteria Tells Advanced Revelation which records from the file to display. For example, you can display records that contain specific values, such as the value "CA" in a STATE field. For details about *selection_criteria*, see "Using Filtering" in the "Introduction to R/LIST" chapter of the *R/LIST* section.

sort_criteria Sets criteria for the sort order of the report. You can sort your data by any number of fields, in ascending or descending order. For details about *sort_criteria*, see "Using Filtering" in the "Introduction to R/LIST" chapter of the *R/LIST* section.

For example, you could sort records by the values in a field that holds zip codes. The records are displayed in numerical order by zip code. Or, you could sort your records by the values in a field that holds company names. The records are displayed in alphabetical order by company name.

fields Tells Advanced Revelation which fields in the file to display. You can choose any of the fields in any order.

keywords Provides a variety of options for fine-tuning your report's format. For example, you can define custom headings and footings to print at the top and bottom of each page. For details on the use of these keywords, see the "R/LIST Keyword Reference" chapter of the *R/LIST* section.

Using EasyWriter at the RList Windows

You can type your LIST statement in the space provided, or you can call the EasyWriter report writer to write the LIST statement. To call EasyWriter, press [F2].

For a complete discussion of EasyWriter, see the "Writing Reports with EasyWriter" chapter.

Testing an R/LIST Report Template

To test your current R/LIST report template, press [Shift-F1] (Test Run). A message window will display. You can save the current report definition before running the test by entering Y at the message window. If you do not want to save the current report template yet, enter N.

Advanced Revelation then displays the report at the terminal. If View is active, the report will display at the View window. (For information about using the View window, press [F1] for online help or see the "Creating Reports with Advanced Revelation" chapter.) When the printing is complete, press [Esc] to return to the Make an R/LIST Report window.

Printing R/LIST Reports

To print a report using an R/LIST report template, from the Main menu choose **Access, Report, and RList**. Advanced Revelation displays the RList window.

A description of the prompts on this window follows. To use a filter to select the records printed on your report, press the Filter key [Ctrl-F10]. For more information on creating and using filters, see the "Using the Filter Key" chapter in *Accessing Data*.

After you fill in the RList window, press [F9] to begin printing the specified report.

- File name** Enter the name of the file that holds the R/LIST report definition. By default, Advanced Revelation stores R/LIST definitions in the **QUERIES** file. To accept the default, press [Enter] at the prompt. If your report definition is stored in another file, enter the name of the file. For a list of available files, press [F2].
- Record** Enter the name of the report definition. For a list of available reports, press [F2].
- Option** Use *Option* to direct your report to the terminal (computer screen) or the printer. By default, Advanced Revelation directs the report to the terminal. These are the valid entries:

Option	Meaning
P	Advanced Revelation prints your report on the printer
T	Advanced Revelation displays your report at the terminal

Chapter H 5

Writing Reports with EasyWriter

This chapter explains how to create R/LIST reports and how to select, sort, and manage lists of records using EasyWriter, a simple-to-use, popup-driven tool.

Introducing EasyWriter	H 5 . 1
Components of EasyWriter	H 5 . 5
Developer Notes for EasyWriter	H 5 . 14



Introducing EasyWriter

EasyWriter is an easy-to-use, popup-driven utility for producing reports using Advanced Revelation's R/LIST report language. EasyWriter provides almost every capability inherent in R/LIST, yet frees you from having to remember the keywords and syntax of the report language.

EasyWriter is especially useful to novice users since it guides them through the entire report writing process in a matter of minutes. Advanced users will also appreciate EasyWriter, because everything needed to create a report — from the file and dictionary names to the R/LIST keywords — is available by making choices from popup windows.

EasyWriter is more than a simple tool, however. Not only does it provide a convenient way for you to create reports, but it includes a variety of report management tools. EasyWriter can recall and re-execute any EasyWriter command you have recently entered. In addition, EasyWriter maintains a full report library, which permits you to create and store report commands you need often. Finally, EasyWriter can import R/LIST commands created earlier by you or other users, and modify, re-execute, and store them.

EasyWriter and R/LIST

EasyWriter actually builds and executes R/LIST commands. R/LIST is Advanced Revelation's ad-hoc retrieval language. Using EasyWriter and R/LIST, you can produce columnar reports from any file currently available.

For instance, you might create a report of all customers currently on file. When you create your report, it might look like this:

[Report Viewer]				
Customer report for Western Region				
Report created on 01 MAY 1988				
CUST.NO	COMPANY	CITY	ST	PHONE
104	ARLINGTON RADIO SUPP	ARLINGTON	CA	(818) 555-1212
826	BOBS SOFTWARE KINGDO	PIEDMONT	CA	(415) 412-8453
106	QUANDO'S BOAT HOUSE	RUSTON	CA	(303) 765-1854
288	SMART-COMM	SAN JOSE	CA	(408) 312-3533

773	PARK PLACE RESTURANT	BELLEVUE	WA	(206) 641-6234
612	ADVANCED GRAPHICS	KIRKLAND	WA	(206) 641-9876
998	ACM ACCOUNTING	KIRKLAND	WA	(206) 821-2316
102	ALLIED SOFTWARE	MALAGA	WA	(415) 555-1212

Page 1				

In this report, each record in the file is presented as a row. Different fields from each record — the customer's name, address, city and phone number — are presented as columns. A heading appears at the top of the page, and a footing at the bottom displays the current page number.

By harnessing the power of R/LIST, EasyWriter provides you with the capabilities you require to produce almost any report from an Advanced Revelation file. EasyWriter offers you these reporting options:

- Create a report from any file currently available to your system.
- Display any fields from the file in your report.
- Order (sort) the report in any way you like. You can sort the report by multiple fields (for example, first by state, and within like states, by city). In addition, you can specify that sorts be ascending (for instance, A to Z) or descending (Z to A).
- Select any records out of the file to appear in the report. For example, you can specify that only those records are to be displayed in which the state is "Florida," or those in which the invoice balance is greater than \$1,000.
- Create custom headings and footings for your report. You can include the current date, time, page number and more in your heading.
- Create running totals and subtotals or averages for numeric data.
- Save your command to be used later. EasyWriter maintains a report library to store commands you need repeatedly.

Each of these capabilities is discussed in more detail later in this section.

EasyWriter and Select Lists

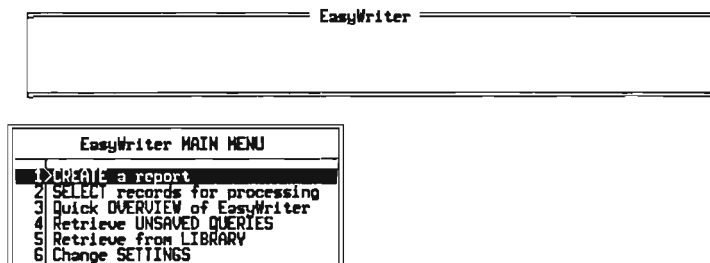
In addition to creating columnar reports with EasyWriter, you can create a select list – a list of records to be used by another process. For example, you can use EasyWriter to create a list of all customers with a balance due greater than \$100.

Not only can you create this list of records using EasyWriter, but you can store it away as well. Later you can activate the list. The list can then serve as the basis for mailing labels, statements, or some other process, including an EasyWriter report.

Invoking EasyWriter

To call up EasyWriter through the Advanced Revelation menu system, start at the Advanced Revelation Main menu. Choose **Design**, and then **Reports**. EasyWriter appears as a choice on this menu. To invoke EasyWriter at TCL (The Command Level), enter EW at the TCL prompt.

When EasyWriter starts up, you will see this menu and window:



On the left is the EasyWriter Main menu. At the top of the screen is the EasyWriter window. As you create a report, the EasyWriter window always displays the R/LIST command you are building. This enables you to see what you have created so far, and what effect your choices have on the command.

EasyWriter at the Filter, Select and Make R/LIST Windows

At these windows you can use the [F2] (Options) key to call EasyWriter for help to write a SELECT or LIST command:

Window	Press [F2]
Filter	at <i>Selection</i> prompt
Select	at window
Make R/LIST	after entry at <i>Template name</i> prompt

Create the SELECT or LIST command at the EasyWriter window and then press [Esc] to return the command to the calling window (that is, the Filter, Select, or Make R/List window). When the current command displays at the calling window, press [F9] (Save) to execute that window process.

The Report Library

EasyWriter maintains a report library of the LIST commands you have built. Anytime you build a LIST command, you can store it away in the library, to be recalled at another time.

This capability allows you to create a command once, and then store it away. Rather than recreating the command each time you require it, you can simply retrieve it from the library. You can edit and re-execute the command, and even store an edited version of it in the library. EasyWriter maintains the library of LIST commands as records in the QUERIES file.

When you store a command in the Report Library, you do not store the full report itself. Instead, you store the R/LIST command that generated your report. When you retrieve and re-execute the command at another time, the report it creates is generated from scratch, using the most current data.

Getting Help

Help is available throughout EasyWriter. Press [F1] (Context help) at any popup to learn more about your current options. For example, to learn more about the choice **Retrieve from LIBRARY**, press [F1] at the EasyWriter Main menu. A message appears with information about each of the functions listed on the Main menu.

Press [Ctrl-F2] (Concept help) to see a message that summarizes the process (selecting, sorting, etc.) you are currently constructing. Concept help is designed to give you an overview of the process, explaining its philosophy and purpose.

In addition to these help options, an explanation of the general functions of EasyWriter appears later in this chapter under the topic "Components of EasyWriter." This section explains the major components of the report writer such as displaying fields, selecting records, etc., and how these functions relate to the popup windows that constitute EasyWriter.

Automatic Help

The option **Set "EASY" MODE ON** from the Change SETTINGS popup enables you to set a Help level. If you set the Help level to Novice mode, explanatory messages will appear as you work through the components of EasyWriter. Each time you enter a new function, a message will explain the function to you, and remind you about keystrokes, etc., that are applicable there.

We recommend that you set Easy Mode ON until you are comfortable with EasyWriter. When you have experience with the report generator, you can set Easy Mode OFF so that the messages do not automatically appear. You will still be able to display them if you need them however, by pressing [Ctrl-F2] (Concept help).

Using Popup Windows

In order to use EasyWriter effectively, you will need to know how to use popup windows. EasyWriter uses four types of popups.

- | | |
|-------------------------|---|
| Single Selection | This popup enables you to choose a single option and return. To use this type, highlight your choice, and then press [F9] or [Enter]. |
| Multiselection | This type enables you to choose one or more options and return. Highlight an option, and press [Enter] to tag it. Tag all the options you wish, then press [F9] to return the selections. |

Sort This type of popup allows you to reorder elements. For example, this allows you to rearrange the order in which fields appear left-to-right on the report. To use this type, highlight the element to move, and press [Enter] to tag it. Then move the highlight bar to the proper location in the popup, and press [Enter]. The tagged item will be moved to the location you highlighted. Press [F9] to save the new order of elements.

Toggle This type of popup alternates or toggles between two choices. For instance, a toggle popup enables you to choose between ascending and descending sorts. To toggle an option, highlight it and press [Enter]. To toggle to the previous value, press [Enter] again. Press [F9] to save your specifications.

To highlight an option in a popup, use the cursor keys to move up and down. You can also enter the number of the choice you wish to highlight.

If you know the text of the option you wish to highlight, you can type in the first character(s) of the word, and then press [Enter]. For example, if you type in "Ret" at the EasyWriter Main menu and press [Enter], the selection **Retrieve UN-
SAVED QUERIES** will be highlighted. If you do this, be attentive to what you type, because the search function is case-sensitive.

Anytime that you want to cancel your choice, or to back up in a series of popups, press [Esc]. This exits the current process without saving any changes, and returns you to where you were when you called the popup.

Components of EasyWriter

EasyWriter works by providing you with a series of popups and windows. These guide you through the process of constructing the R/LIST command for your report or filter. Each component of the command or filter – the selection (comparison) criteria, sort criteria, display criteria, headings, footings, and modifiers – is comprised of a series of these popups and windows. You simply need to choose the option(s) you want from the popups as they appear, and fill in data at the windows as prompted.

Remember that the online Help system will provide you with details about the popup or window you are using. Press [F1] (Context help) to see specifics about the popup options that are available. Press [Ctrl-F2] (Concept help) to read more about the purpose of the process you are using.

The Current Report

For most EasyWriter processes, you will be creating, modifying, or saving a current report. For instance, the option **CREATE a Report** walks you through the process of creating an R/LIST command. As soon as you choose this option from the EasyWriter Main menu, you will begin constructing a new report. That new report will then be the current report.

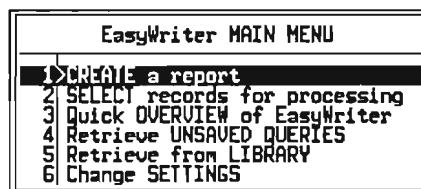
On the other hand, if you choose the option **Retrieve from LIBRARY**, you will be recalling a report created earlier. As soon as you select an item from the Report Library, that item becomes the current report.

When you are at the EasyWriter Main menu, there is no current report. Because of this, there is no option at the Main menu to add a report to the library, or to modify a report. In order to access these functions, you must first retrieve or create a report to serve as the current report.

Conversely, if you are editing the current report, and attempt to return to the Main menu, you will see a message warning you that if you return to the Main menu the current report will be cleared. If you proceed, the current report is cleared and you have the opportunity from the Main menu of creating or retrieving a different report.

The Main Menu

The EasyWriter Main menu is the entry point into the six basic functions you can access. The Main menu looks like this:



CREATE a Report

When you choose this option, the end result will be a columnar report that displays on the screen or is printed on the printer. You will have the opportunity to specify six components of the report:

- Which file you wish to report on
- The fields that are to be displayed
- The order in which the records are to appear
- Which records are to appear

- Headings, footings, and other formatting specifications, including subtotals and averages
- Whether the report displays on the screen or prints to the printer

The word **LIST** will appear in the EasyWriter window, as will a popup displaying all the files available to EasyWriter. You must select the file for which you are creating a report. Once you have done so, you may construct the rest of the components of the command using popups.

SELECT Records

The result of this option is not a report, but a list of record keys. Use this option when you wish to set aside a list of records to be used in another process.

For example, you might have a process that prints mailing labels. You can use the **SELECT** option to create a list of all customers in a specific area of the country. You can save the list, and then later retrieve the list and use the Advanced Revelation Label processor to create mailing labels.

The **SELECT records** option, like the **CREATE a Report** option, permits you to specify which records to select, and in what order. Because you are not creating a report, you do not need to specify display fields or formatting options, or to indicate screen/printer output.

When you first choose this option, the word **SELECT** appears in the EasyWriter window. You will also see a popup of available files. You must choose the file from which you will be selecting records. Once you have done so, you can specify the rest of the options (which records, what order) by using the popups.

UNSAVED Queries

This option is used to retrieve and re-execute the most recent reports or record key selections. You can use this option even if you did not save your report command in the Report Library.

Advanced Revelation always remembers a certain number of your most recent commands (the exact number depends on your user environment). EasyWriter can look through these recent commands and pick out the ones that generated a report or a record key list.

Retrieve from LIBRARY

Anytime that you create a report or a selection command, you can store the command in the Report Library. This option from the EasyWriter Main menu allows you to retrieve stored commands.

In order to store the command in the Report Library, you must use the option **ADD report to LIBRARY**. This option appears on both the **REPORT** options and the **SELECT** options popups. To use the option, first create the report or selection command, and then choose the **ADD report to LIBRARY** option to store the command.

Change SETTINGS

EasyWriter can be configured to suit your environment, your application, and your level of expertise.

Environmental settings include the size of the screen and printer page. For example, if your printer can only print 80 columns across a page, you can use this option to make EasyWriter aware of this.

Developers can control the files that are available to EasyWriter. Details on this capability appear in "Developer Notes for Easywriter" in this chapter.

Finally, you can set a Help level for EasyWriter. If Easy Mode is set to ON, EasyWriter will display help messages as you move from popup to popup. The messages inform you about the popups you are using, and remind you about important keystrokes. When you are familiar with EasyWriter, you can set Easy Mode to OFF so that these messages do not appear automatically.

Specifying WHICH RECORDS

This option, which is called from both the CREATE a Report process as well as the SELECT Records process, is used to tell EasyWriter what records you wish to use in your report or in your record key list. In order to do this, you must create one or more comparison clauses.

The Comparison Clause

A comparison clause determines which records will appear in the report. The clause compares the value of a field against a fixed value or against the value of a different field in the record. If the comparison for the current record results in a match, the record is included in the report. If not, the record is not displayed.

For example, you may wish to print a report displaying all customers in the state of California. The comparison clause will then look into the state field of each record. If CA appears in the field, the record is displayed.

A comparison clause consists of at least these elements:

- A field to base the comparison on
- A comparison operator, such as = (equals), > (greater than), etc.
- A value or field to compare against the data in the original field

To use the above example, the comparison clause to select California customers might read something like:

```
ST EQ "CA"
```

The field to base the comparison on is ST (state). The comparison operator is EQ (equal), because you want to find records in which the state is (equals) California. Finally, the value to compare to is CA.

Creating a Comparison Clause

To create a comparison clause, you will step through a series of popups. Assume for a moment that you are creating a report based on the CUSTOMER file. The sequence of popups will be:

CUSTOMER Fields (choose the original field)



COMPARE Words (choose the operator =, >, etc.)



CUSTOMER Fields (choose the comparison word or field)

From these popups, you can construct the comparison clause ST = "CA" or something similar.

Multiple Comparisons (ADD a COMPARISON)

You can expand the comparison clause to compare the contents of a field against more than one value. For example, perhaps your report is to be for customers in California and in Nevada. In that case, the comparison clause would read:

ST = "CA" OR EQ "NV"

This causes the report to select all records in which the state field contains either "CA" or "NV" – either is acceptable.

Change the Relationship between Comparisons

When you add an additional comparison value, it is connected to the earlier values with an OR. This means that if any one of the values in the comparison clause appears in the field, the record is displayed.

There will be instances in which this is not appropriate. For example, you may wish to find all customers in which the company name contains the words ACME and INTERNATIONAL. In this case, both words have to appear. If only one of these words appears in the company name, the record is not acceptable.

To create this type of comparison clause, create a clause with multiple comparison values. Then change the relationship between the comparisons. The option **Change RELATIONSHIPS between comparisons** allows you to change the word OR to AND. This changes the comparison so that all the values in it must appear in the field.

Multiple Comparison Clauses

Sometimes the rules for displaying records involve more than one field. For example, you may wish to create a report listing all 1988 invoices for your Florida customers. The comparison clauses to generate this report examine two fields: ST and INVOICE.DATE.

If you examine more than one field, you must create more than one comparison clause. In this instance, you create the first comparison clause based on the ST field:

```
ST EQ "FL"
```

A second comparison clause is based on the INVOICE.DATE field:

```
INVOICE.DATE FROM "01-01-88" TO "12-31-88"
```

The default relationship between multiple comparison clauses is that either criterion is acceptable (an OR is placed between the clauses). In this example, however, because both criteria must be met for a record to be displayed, you must change the relationship between the separate clauses so that there is an AND between them.

There is no limit to the complexity of the rules for displaying records. You can have any number of clauses, each of which examines any number of values. You can establish any relationship you require using AND and OR to connect the values or clauses.

Including vs. Excluding Records

When a comparison clause has been established, a report compares the contents of a field against a fixed value or against another field. If the record fulfills the criterion in the comparison clause, it is included in the report.

You can also create comparison clauses that cause records to be excluded from the report. For example, you might create a report that displays all customers except those whose status is "Inactive."

You can use the **Change comparisons to INCLUDE/EXCLUDE records** option of the COMPARISON Options popup to change between inclusive and exclusive comparison clauses. If a clause is inclusive, it begins with the word "WITH." An example is:

```
WITH ST EQ "CA"
```

An exclusive clause begins with the word "WITHOUT." An example is:

```
WITHOUT STATUS = "Inactive"
```

Selecting Specific Records

If you know the record key or keys of the records you wish to display, you can use the **CHOOSE records using SPECIFIC KEYS** option from the COMPARISON Options

popup to enter these. This will result in a faster report than if you base your comparison on a non-key field.

For example, if you are creating a report of certain employees, you could base the report on the employees' names. However, the report will run faster if you base it instead on the field you use to identify the employees in the file (the key field). This might be the employees' social security numbers or their employee numbers.

Specifying Record Sort Order

Records in a file are not maintained in any particular order. If you wish to display them in a specific order, you must order (sort) them in the report.

The option **Specify record SORT ORDER** allows you to do this. Sort order is based on the contents of fields in the record. For example, a report of customers might be sorted by the contents of the ZIP field, the COMPANY field, or some other field.

You can sort by any number of fields. If you specify a second sort field, it will be applicable to records that have the first sort field in common. For example, if your first sort is by ZIP code, and your second one by company, the report will appear in ZIP code order. If there is more than one record with the same ZIP code, that group of records will appear in company order. Third, fourth, etc. sort fields are applied in a similar hierarchy.

Ascending and Descending Sorts

Unless you specify otherwise, sorts are always ascending. Records are sorted from lowest to highest value. Records with character data will appear in alphabetic order, and records with numeric data in numeric order.

If you need to, you can change one or more sort fields to use descending order. For example, if you want a report with the most recent date first, you can specify a descending order for the date field.

Displaying Records

In addition to simply displaying a field in a report, you have several options for modifying the display. These include reformatting a field, totalling or averaging a numeric field, and grouping fields that have been sorted.

Reformatting a Field

When you display a field in a report, the display characteristics of that field are read from the dictionary of the current file. These characteristics include the display length, the justification of the field (left, right, center or text (paragraph)), the column heading, and the display type (conversion). If you reformat a field, the formatting specifications are in effect only for the current report.

For example, you might decide that you would like to display the COMPANY_NAME field. As defined in the dictionary, it might be 25 characters wide. However, you decide that since you don't have much room on the report, you can get by with only 20 characters of the company name.

To temporarily change the display length of the COMPANY_NAME field, choose the **REFORMAT a display field** option from the DISPLAY Options popup. This displays a window with prompts for the four format specifications you can change. Choose the field to reformat, and the window will appear with the existing (default) values for these specifications filled in. You can change any of the four values. Use the [F2] key to see options for *Justification* and *Output Format*.

Total and Average

If you have fields with numeric data in them, you can have a total or an average of those fields printed at the end of the report. Numeric fields are those defined in the dictionary with a display type of numeric (MD).

If you also group the fields you are totalling or averaging, you can get a subtotal or subaverage printed at each break in the report. A final total or average then appears at the end of the report.

Grouping Records

If you are sorting (ordering) your report and displaying the fields on which you are basing the sort, you can choose to group those records. When you group records, the report shows a "break" when the value of the sort field changes.

For example, if you create a customer list sorted by zip code, you might also group the report by zip code. Whenever a new zip code displays on the report, a break in the report — some blank lines, or even a new page — is generated in the report. Because grouping causes a break in the report whenever a new value is encountered, it is also called "breaking on" a value.

The normal break in a report is a blank line, a break indicator, and then another blank line. The break indicator is normally *** (three asterisks). Options available from the GROUP (BREAK-ON) ordered values popup allow you to substitute different break indicators.

Alternatives to the three stars include starting a new page, printing the value of the group field, or including text that you supply. If you are also totalling the field, a subtotal will appear at each break. You can specify options that display an underline and an overline for the subtotal. You can even have the value of the group field appear in the heading — this is useful if you are also specifying that each new group be put on a separate page. If you choose this option, you must choose a similar option for the heading.

Heading and Footing

A heading is text that appears at the top of every page of the report. A footing appears at the bottom of each page.

A heading or footing consists of text that you provide. Options in the HEADING window and in the FOOTING window enable you to include additional information. For example, you can display the current date, time, page number and file name in addition to your own text. Press [F2] at the HEADING and FOOTING windows to see a list of these options.

You can also include the value of a field for which you have specified the group option. To do so, you must specify this same option in the GROUP (BREAK-ON) options popup.

Other Formatting Options

Whenever EasyWriter creates a report, it makes certain assumptions about the format of that report. Unless you specify otherwise, the report will always:

- Display a heading
- Display the record key
- Display column headings
- Display all detail records when generating totals
- Single-space the report

The Special **FORMATTING** options popup enables you to change these defaults. For example, you can specify that detail records are not to be displayed when you are generating subtotals. If you do so, the report will consist of subtotals and a total only — the individual records from which the totals are calculated will not display.

Printer and Screen

Reports are normally displayed on the screen. If you wish to route a report to the printer, you must choose the **Reroute to PRINTER** option from the REPORT options popup.

Once you have chosen this option, the popup changes. The option will change to read **Reroute to SCREEN**. This permits you to change your mind and reroute the report back to the screen.

ACTIVE KEY List

Lists of active keys are typically used when you wish to select and sort a set of records, and then use that list of records several times. If you generate the list once and reuse it many times, you can avoid selecting and sorting the records each time you need them.

To create a list of active keys, use the **SELECT Records for processing** option from the EasyWriter Main menu. This enables you to select and sort a list of records. When the list is assembled you can use the **Change ACTIVE KEY list** option to save the keys away.

The same option enables you to retrieve the keys later. When you do so, the list of active keys becomes the basis for further processing. For example, if you create a report, and then activate a list of keys representing a subset of a file, the report will only display records whose keys are on the list.

The advantage is that you do not need to select and sort records each time you run the report. You can select and sort the records one time, and then store the list of records away. You can then activate the list whenever you like, and run a report against it. The report does not need to include selection and sort criteria, since these have already been incorporated into the list of active keys.

If you activate a list of keys, but then decide that you do not need that subset of records, you must clear the list of active keys. An option from the Change ACTIVE KEYS options popup enables you to do this.

Developer Notes for EasyWriter

This section provides details about the use of EasyWriter. Included is information about calling the EasyWriter as a subroutine and information about customizing the files that appear for an EasyWriter user. In addition, there is information about features of R/LIST not used by EasyWriter.

Calling EasyWriter as a Subroutine

You can call EasyWriter as a subroutine directly. This enables you to invoke EasyWriter with a code and command, or to call it from within your own application. When you call EasyWriter as a subroutine, it assumes that you wish to create a report (LIST).

To call EasyWriter, call the subroutine EW.VIEWER. This will invoke EasyWriter directly, bypassing the Main menu. You can pass parameters to EasyWriter in one of two ways.

The first is to pre-load @PSEUDO with a LIST statement. For instance, you might load the command LIST plus a file name into @PSEUDO. This will then be parsed by EasyWriter, and the popup for a file name will be skipped. This method can be used, for instance, to substitute EasyWriter for the default R/LIST window used by the [Alt-V] key in windows.

The second method to pass data to EW.VIEWER is to load the file name into the AREV.COMMON variable DATAFILE. EW.VIEWER will examine @PSEUDO first. If nothing appears there, it will attempt to read the current file name from DATAFILE.

Customizing the EasyWriter Files Popup

Whenever a new report or select is begun, a popup of available files is presented to the user. This popup can be customized to exclude certain files, and to present the file in a specific order. These customizations are available from the **Change SETTINGS** option of the EasyWriter Main menu.

Excluding Files The option **Specify WHICH FILES to exclude** presents a toggle popup of files. Highlight and toggle to "Yes" the files you wish to exclude from the EasyWriter Files popup. These would typically be dictionaries and system files, but can also include files to which you wish to limit access.

This popup builds a record in the SYSTEM file with a key of *username.EXCLUDE.FILES*.

Ordering Files The option **Specify FILE DISPLAY ORDER** presents an ordered popup of the files remaining for display. The order of the files can be changed, so that the most important or most often used files can appear at the top of the EasyWriter Files popup.

This popup builds a record in the SYSTEM file with a key of *username.FILE.PRIORITY*.

Customizing the Change SETTINGS Popup

You may wish to remove the two file display options from the EasyWriter Change SETTINGS popup. By doing so you can prevent users from changing the EasyWriter Files popup from the way you have configured it.

This popup is stored in the SYS.POPUPS file under the name EW.CONFIGURE.OPTIONS. You can edit it as a standard popup using the Make Popup window. Remove the option(s) that you do not want to display.

Importing R/LIST commands

EasyWriter can import almost any LIST or SELECT command already available. Commands are imported into EasyWriter by executing them and retrieving them from the TCL stack (the **Retrieve UNSAVED QUERIES** option). You can also import a command by placing it into the QUERIES file directly, and then using the **Retrieve from LIBRARY** option. Commands imported in this manner can be edited, re-executed, and stored by EasyWriter.

There are certain features of R/LIST, however, that EasyWriter cannot use. If EasyWriter encounters one of these while importing a command, a message appears stating this fact, and the import process is halted.

R/LIST features that EasyWriter cannot use are:

- NOT
- EVERY
- LIMIT
- Parentheses
- DICT override

Chapter H 6

Creating Merge Reports

This chapter explains how to create Merge reports that combine formatted text with data from a file.

Introducing Advanced Revelation's Merge Processor . . .	H 6 . 1
Creating a Merge Template	H 6 . 2
Entering Scripts	H 6 . 7
Executing a Merge	H 6 . 19
Special Features of the Merge Processor	H 6 . 21



Introducing Advanced Revelation's Merge Processor

The Merge processor provides a full-featured, flexible method of merging data from an Advanced Revelation file into a form or document that also includes literal text, variable information such as date and time, headings and footings, and even special text such as personalized or conditional messages.

In addition to its basic capabilities, the Merge processor includes a variety of advanced features. These include individual paragraph formatting to create left justified, filled and unformatted paragraphs, and multipage capabilities such as widow and orphan control and control for variable-length tables. Character formatting includes underlining, boldface, and other desirable features. These can be tied to individual printers using the Merge processor's printer setup facility.

Finally, the Merge processor is able to accommodate unusual or complex requirements. Portions of a Merge template can be made conditional, and many points in the Merge processor are open-ended to accept developer customization.

Merge Processor vs. Form

Advanced Revelation already includes a forms processing capability in its Form processor. However, there is a difference in the use of these two processes.

The Form processor is suited to creating forms that you can paint out on the screen. This is particularly useful in situations where you wish to merge data from an Advanced Revelation file with a preprinted form or similar fixed format. In general, the Form processor is used to produce output that looks exactly like the template on the screen.

The Merge processor, on the other hand, is better suited to creating forms for variable data. Unlike the Form processor, the Merge processor is expressly designed to accept and format varying quantities of data. This includes the ability to merge and then format data into paragraphs, and to flow text and data onto multiple pages if necessary.

Fundamentals of the Merge Processor

The Merge processor works something like the Label processor. In essence, you create a Merge template that includes text for the body, data from a file, plus headings, footings and control information such as data file name, page size, margins, etc. To do this, you fill in prompts in a special window designed to help you create Merge templates.

When the Merge template is done, you execute a command that reads the template and merges data from the data file into the template. The result is a series of finished documents or reports that can be displayed on the screen, printed on a printer, or written to a file.

The Merge template consists of the control information for the document as a whole, plus the basic text, called the *script*. Variable information such as data from the file or character or paragraph control information is included in the text using special sequences called flags. For example, the following text illustrates the use of a flag to include data from a field in the data file:

Dear {CONTACT.NAME} :

In this example, data from the field CONTACT.NAME will be inserted into the text at this point.

Control information is embedded in a similar fashion. The following illustrates the use of flags to cause portions of the text to be put into italics:

send your payment «ITAL»today«END.ITAL»!

In the example, the word “today” will appear italicized in the final output.

Creating a Merge Template

The first step in executing a merge process is to create a Merge template. The Merge template contains information such as the basic format of the document, the type of printer to be used to print the document, and the script itself.

The Make Merge Window

To create a Merge template, use the Make Merge window. The Make Merge window is available by using the **Design-Report-Merge** option from the Advanced Revelation Main menu, or by entering this command at TCL:

CW MAKEMERGE

When you invoke the Make Merge window, a two-page window will display. The first page looks like this:

Make Merge									
Merge Name									
=FILES									
Data		Special Text				Output			
=MARGINS									
Top	Bottom	Left	Right	Para Indent	PARA CONTROL		PAGE DIMENSION		
					Widow	Orphan	Width	Height	
Default Format					Printer				
=HEADER									
=FOOTER									

The second page is a single prompt called *Merge Script*.

The prompts in the window are grouped by function. For example, all prompts concerning margins appear together under the label **MARGINS**.

Fill in the prompts in the window to create the Merge template. Then press [F9] to save the template in the MERGES file. Be sure to use [F1] (Context Help) if you are unclear about the purpose of a prompt. Many prompts also provide options if you press the [F2] Options key.

Make Merge Window Menu

The Make Merge window has its own menu, available when you press [F10]. The menu includes options to test scripts, do a test run of the script, and import scripts from an operating system file. The test options on this menu are discussed later in this chapter under “Executing a Merge”. Other features of the menu are described under “Special Features of the Merge Processor”.

Make Merge Window Prompts

The prompts in the Make Merge window enable you to establish the Merge script itself, as well as to enter a variety of control information about the Merge template.

- | | |
|---------------------|---|
| Merge Name | Whenever you create a Merge template, you give it a unique name. The template is filed away using this name, and you can recall it anytime later.

Enter any name you wish. The Merge template name can be up to 50 characters long. You can use any characters except spaces in the Merge template name. If you wish to see a list of templates already defined, press [F2]. |
| Data | Enter at this prompt the name of the file containing the data you wish to merge with the template. For example, if you wish to merge this Merge template with data from the CUSTOMER file, enter CUSTOMER at this prompt. Press [F2] to see a popup of all available files. |
| Special Text | If you will want to merge special text such as personalized notes or postscripts with the document, enter here the name of the file that contains that text. Press [F2] to see a list of files available.

For more information about the use of Special Text, see “Special Features of the Merge Processor” later in this chapter. |

Output If you enter the name of a file at this prompt, you can have the Merge process route its output to this file. This capability is used in order to create an entire file of merged documents preparatory to printing them. For more information on the use of the Output file, see “Executing a Merge” later in this chapter.

Top This is the number of rows left blank at the top of the page. For example, to leave a one-inch margin at the top using a standard 6 lines-per-inch printer, enter 6 at this prompt.

The heading text (see below) is printed in the top margin, so be sure to provide enough space for the heading you create.

Bottom Enter at this prompt the number of blank lines to be left at the bottom of the page. For example, to leave a one-half-inch margin at the bottom of the page using a standard 6 lines-per-inch printer, enter 3 at this prompt.

The Footing text (see below) is printed in the bottom margin, so make sure that you are allocating enough space for it.

Left This is the number of columns to be left blank on the left side of the page. This number represents the number of characters in the current pitch (10 pitch = 10 characters per inch).

For example, if you wish to leave one inch blank on the left side of the page, and your printer is set to print 10 characters per inch, enter 10 at this prompt.

Right This is the number of columns that will be left blank on the right side of the page. This number is calculated in the same manner as the left margin.

The right margin represents the minimum amount of space left blank on the right side of the page. If you format paragraphs using left justification, or no justification at all, text will appear “right-ragged,” with each line ending at a different point in the page. None of the lines, however, will run farther than the right margin.

Para Indent The number entered at this prompt is used as the default number of columns by which the first line of each paragraph is indented. For example, if you enter 5 at this prompt, the first line of each paragraph in the document will be indented five characters.

This number can be positive or negative. If the number is positive, an indented paragraph is created. If the number is negative, the first line of the paragraph is flush with the left margin, and the remainder of the paragraph is indented. For example, if you enter -5 at this prompt, all lines in a paragraph except the first one will be indented five lines from the left margin.

Widow This prompt controls the minimum size of a paragraph that will appear at the bottom of a page. This feature is used to avoid having lines from paragraphs “widowed” when the bulk of a paragraph appears on the next page.

The number entered at this prompt defines the size of a “widow”. Anytime that a paragraph splits between pages and leaves this many lines or fewer, it will be reformatted to fit onto the next page.

Orphan An “orphan” is the opposite of the widow described just above. An orphan appears at the top of a page.

The number of lines entered at this prompt defines an orphan. If a paragraph splits and flows this many or fewer lines to the next page, it will be reformatted to fit onto the next page.

Width This number defines the total size of the page in columns. For example, a typical letter-sized piece of paper is 8-1/2" wide. You would therefore enter 85 at this prompt.

The number entered at the *Width* prompt is used in conjunction with the left and right margins to establish the size of the area in which text can appear. For instance, a page 85 columns wide less a 10-column left margin and a 10-column right margin leaves 65 columns in which to print text.

Height The number entered here is the total length of the page in rows. For example, a typical letter-sized page is 66 lines long.

This number, less the space for the top and bottom margins, determines the size of the area in which the body of the text will appear.

Default Format Each paragraph in the document can have its own format. However, the entry at this prompt determines the format for paragraphs having no special format.

The Merge processor recognizes three types of paragraph formatting: left, filled, and unformatted. Press [F2] to see a popup of formatting options. The choices are:

Format	Result
LEFT	The left margin is aligned, but the right margin is ragged.
FILLED	The paragraph is padded with spaces so that each line is aligned at both the left and right margins.
NOFORMAT	No attempt is made to format, and text is output as it appears in the script, except for formatting changes.

Printer Enter at this prompt the name of a printer that has been defined for the Merge processor. The Merge processor uses the printer type to translate generic codes for character formatting (such as bold, italic, etc.) into specific codes for your printer.

Press [F2] to see a popup of printer types already defined. If you wish to create new printer types, use the **Printer** option from the Make Merge menu.

For more information on defining and using printers, see “Defining Printers for the Merge Processor” later in this chapter.

Header Use this prompt to enter text that will appear at the top of each page of the document. The heading can be as long as the top margin. Be sure that you have allocated enough space in the top margin to accommodate the heading.

To enter text for the heading, begin typing at the prompt, or press [F2]. In either case, a Merge Header window will display, allowing you to enter or edit the heading text.

Enter text as you would in any window. The heading text is treated as a normal script, so all page and paragraph formatting is applicable. You can also include all types of flags, such as data fields or flags for paragraph or character formatting.

For more information on field, paragraph and character flags, see “Entering Scripts” later in this chapter.

When you are done entering heading text, press [F9] to save the heading. Press [Esc] to return to the Make Merge window.

Footer Use this prompt to define up to three lines of text that appear at the bottom of each page of the document. You can enter and edit footing text exactly like heading text (described just above).

Merge Script This prompt appears by itself on the second page of the Make Merge window. It is used to enter the body text of the document.

As with the heading and footing, entering or editing the script is done in a separate window. When you begin typing, or if you press [F2], the Merge Script window will appear. All script entry and editing is done in this window.

Enter the text of the script in the Merge Script window. You can embed flags in the script in order to control paragraph and page formatting. For more information on the use of flags, see “Entering Scripts” later in this chapter.

When you are through entering or editing your script, press [F9] to save it. Then press [Esc] to return to the Make Merge window.

In the Make Merge window, press [F9] to save your Merge template.

Entering Scripts

The script is the heart of the Merge document. It contains the text that will appear when the document is merged with data from a file. In addition, it contains information about what data will appear in the document, as well as the format of paragraphs and characters in the final document.

The script consists of literal text, which is to appear exactly as you enter it, except for format changes. The script also contains flags for indicating data fields to merge, as well as flags that specify paragraph, document, and character formats that are to be used in the text. These flags are translated into the appropriate data or codes when the document is actually merged.

Scripts are entered at the *Header*, *Footer*, and *Merge Script* prompts in the Make Merge window. To enter scripts at these prompts, use the cursor keys or the [Enter] key to get to the appropriate prompt, and then press [F2]. This will display a special script entry window. Enter the script at this window. When the script is finished for that prompt, press [F9] to save it, then press [Esc] to return to the Make Merge window. Press [F9] at the Make Merge window to save your Merge template.

While you are editing a script, the [F2] key also provides a popup of all flag options. From this popup you can in turn display other popups of all field, format, and character flags. The popups display the flag itself as well as a brief explanation of the flag’s meaning.

Anytime you wish to insert a flag, press [F2]. Special softkeys enable you to call the appropriate popup (field flags, character flags, etc.) directly.

Entering Text

When the Header, Footer or Merge Script entry window displays, you can enter the script as if you were creating a document using a word processor. Use the cursor keys and the [PgUp] and [PgDn] keys to move around in the script entry window.

- ✎ The script entry windows use the Advanced Revelation full-screen editor. For more information about using this editor, see the chapters called “Entering and Editing Data in Windows” and “Using the Full-Screen Editor” in *Getting Started*.

Enter text as you wish to see it displayed. Except for formatting text (see below), everything you enter will be displayed. Create separate paragraphs of text by placing a blank line between them.

For example, you might enter two paragraphs like this:

To take advantage of this offer, you must act today.

Return the enclosed coupon with your check, money order or credit card number.

The text will appear exactly as you have typed it, except for any formatting you have specified. For instance, if your default format specifies justified text, the paragraphs will appear justified when they are printed or displayed.

Entering Field Flags (Alt-F3)

To incorporate data from a data file, you will need to include field flags in the script. Field flags designate the name of the field from which data will be extracted.

Field Names To include a field flag, press [Alt-F3] while in a script. A popup will appear that displays all fields from the data file for which you are creating the script. Choose the field that is to appear in the script at this point.

For example, to create a customized salutation in a letter, type the word “Dear”, and then press [Alt-F3]. A popup of all fields appears. Select the field that is to appear in the salutation (such as FIRST.NAME). When you press [F9] to save your selection, you will be asked to select a field format. Skip this popup by pressing [Esc]. The following will appear in the script:

```
Dear {FIRST.NAME}
```

The braces ({ }) around the field name indicate that the text inside is the name of a field. When the document is merged, the first name data from the current record replaces the field flag.

Reserved Words

The first four options in the Merge Fields Select popup are for special reserved words. These reserved words allow you to insert flags for the current date, time, and page number into the script.

Reserved word flags are treated like field flags. However, the words inside the braces (for instance, MRG.PAGE) are not replaced by data from the record. Instead, the special function called by that word (for example, the current page number) is merged at that point.

The reserved words used as field flags are:

Word	Meaning
MRG.PAGE	Insert current page number
MRG.TIMEDATE	Insert current time and date
MRG.DATE	Insert current date (internal format)
MRG.TIME	Insert current time (internal format)

- ☛ If you use the flags for date or time (MRG.DATE and MRG.TIME), you will need to format the fields. For details on field formatting, see “Field Formatting” later in this chapter.

Entering Format Flags (Alt-F1)

A special set of flags enables you to indicate various types of paragraph and document formatting. Formatting flags are not surrounded with braces, as are field flags. Instead, they are surrounded with chevrons (« »). To see a list of available format flags, press [Alt-F1] while editing a script.

Formatting flags are divided up into three different types. The first type establishes or changes the format of paragraphs or lines. The second type formats a block of multivalued fields. The third type indicates conditional inclusion of text or other script information.

Paragraph Formatting Flags

Paragraph formatting flags are used to create or override a format for paragraphs or individual lines. Each of the flags has a different scope. Some are document oriented (they work for the document as a whole), while others are limited in scope to a single paragraph or line. See the following table.

Table H6-1
Format Flags for the Merge Processor

Flag	Meaning	Scope
«UNFORMAT»	Turn default format off	Document
«FORMAT»	Re-establish default format	Document
«PARA.INDENT, <i>n</i> »	Establish paragraph indent of <i>n</i> characters	Document
«CENTER»	Center an unformatted paragraph	Paragraph
«END.CENTER»	End centering, effective for the current line	Paragraph
«DBLSPC»	Double space paragraph	Paragraph
«RIGHT»	Right-justify remainder of line, if unformatted paragraph	Line
«PAGE»	Force a page break	N/A
«PARA»	Force a paragraph break, no space	N/A

If the scope of a flag is “Document,” the flag is in effect for the rest of the document unless explicitly turned off. For instance, once the «UNFORMAT» flag is activated, all subsequent paragraphs are unformatted until the «FORMAT» flag is encountered. Flags with a “Paragraph” scope only affect the current paragraph. Line-oriented flags only affect the current line.

A flag such as «RIGHT» or «CENTER» is only effective for an unformatted paragraph. This is because the paragraph format otherwise conflicts with this flag. If there is a conflict between a paragraph format and a flag, the paragraph format takes precedence.

The flag «PARA» is used to create paragraphs with no blank lines between them. Normally, a paragraph boundary is established by putting a blank line between two paragraphs. The «PARA» flag, however, forces a paragraph break without inserting a blank line. Inserting the «UNFORMAT» or «FORMAT» flags into a paragraph also forces a new paragraph at that point.

Multivalued Blocks A special set of flags enables you to define blocks of multivalued fields. Multivalued blocks are used to establish formatting and headings for groups of multivalued fields.

A multivalued block is marked with the flags «VBLOCK» and «END.BLOCK». Inside of this block can appear field flags, literal text, or other standard script information. The information in the block will be repeated once for each value or set of values in the multivalued field. If the fields in the block have different numbers of values, the information is repeated for the maximum number of values in the longest field.

The block is repeated as a whole each time. All text and flags that appear between the start and end of the block – including spacing, carriage returns, etc. – will be repeated each time.

A multivalued block is treated as a normal paragraph, and is subject to any formatting that is applicable to other paragraphs. As a result, spacing, carriage returns, etc., are all formatted as they would be in the current paragraph format.

In order to establish columns, a multivalued block *must* be tagged as an unformatted paragraph. If you have a default paragraph established, you should set the «UNFORMAT» flag before the multivalued block, and reset the «FORMAT» flag afterward.

☞ Do not follow a «VBLOCK» flag with a blank line.

☞ To print values in columns, the «VBLOCK» and «END.BLOCK» flags must be entered on separate lines. Either of these formats will print correctly:

```
«VBLOCK»
{field} {field} «END.BLOCK»
```

```
«VBLOCK»{field} {field}
«END.BLOCK»
```

To print values in columns and use double spacing (a blank line between column rows), enter the «VBLOCK» and «END.BLOCK» flags above and below the literal text and field flags that make up the multivalued block. For example:

```
«VBLOCK»
{field} {field}
«END.BLOCK»
```

You can also designate a multivalued block heading. This heading appears at the top of the multivalued block, and is typically used as a column heading above the multivalued field. If the multivalued block is longer than one page, the heading is repeated on subsequent pages.

To establish a multivalued block heading, use the flag «VBLOCK.HEAD». Everything that appears after this flag and before the «VBLOCK» flag will be used as a heading. As with the block itself, all paragraph formats are applicable.

An example block that includes a heading might look something like this:

```
«UNFORMAT»
«VBLOCK.HEAD» Invoice Date Amount
«VBLOCK» {INVOICES} {INV.DATE} {INV.AMT}
«END.BLOCK»
«FORMAT»
```

In the example, the multivalued block is enclosed with flags that first disable, and then re-establish paragraph formatting. A block heading is established with a fixed number of spaces between the columns. The block itself consists of three field flags with spaces between them. A carriage return follows the block, so that a return will be executed for each value printed.

Conditional Scripts

It is possible to designate the inclusion of text or flags conditionally. For instance, you may wish to display certain fields only if they exist, or only if other fields in the record exists.

The flags «IF», «ELSE» and «ENDIF» are used for conditional blocks. «IF» and «ENDIF» are required; «ELSE» is optional for situations in which you wish to designate alternative text and flags.

The condition is based on whether data appears in one of more fields. The fields to be tested appear in the conditional block as field flags. The field flags can appear anywhere between the «IF» and the «ELSE» or «ENDIF». For instance, the following illustrates the use of a conditional block:

```
«IF»By the way, you are eligible for a
discount of {DISCOUNT} on anything in our
catalog.«ENDIF»
```

In the example, if there is anything in the DISCOUNT field for the current record, the entire block will be printed (including the field DISCOUNT). If there is nothing in the DISCOUNT field, this block of text will be skipped.

An example of an «ELSE» flag:

```
«IF»Dear {FIRST.NAME}:«ELSE»To whom it may
concern:«ENDIF»
```

Here, if a first name exists, the salutation will be “Dear” plus the first name. If no first name is on file, the text “To whom it may concern” is printed.

You can designate that a field is to be used as a condition, but that it is not to be printed. To do this, put ? (question mark) as the first character in the field flag. For example:

```
«IF»{?INVOICES}We note that you have
already placed an order with us.«ENDIF»
```

In the example, if there are any invoices on file for this record, the text is printed. Because the field flag includes ?, it will not be printed; it is only used to test the condition.

Finally, you can base a conditional block on a combination of field flags. If more than one field flag appears in the «IF» block, there must be data in each one of the fields in order for the block to be printed. Multiple fields in the «IF» block are linked with a logical AND.

If you wish to identify one or more field flags as particularly important, you can put * (asterisk, star) at the beginning of the field flag. This serves two functions. The first is that it causes only the field(s) so marked to be considered in testing the condition. All fields in the block are displayed, but unmarked fields are not included in testing the condition. For example:

```
«IF»Dear {*FIRST.NAME} {LAST.NAME}:«ELSE»To
whom it may concern:«ENDIF»
```

In the example, if a first name appears, the “Dear” section of the block is printed. If a last name exists, it will be printed as well, but it is not used as a test for this condition. If no FIRST.NAME is on file for the current record (even if there is a last name), the second half of the block is printed.

The * also changes the logic of testing. If * is used in field flags inside a conditional block, the fields are linked with a logical OR. If any of the marked fields contain data, the block is printed.

Entering Character Flags (Alt-F2)

Character format flags enable you to specify such formats as boldface, italic, underline, or other font characteristics. You can establish any character formats that your printer is capable of producing. To see a popup of available character flags, press [Alt-F2] while editing a script.

Character format flags look much like paragraph format flags. They consist of a keyword (such as *ITAL* for italic) surrounded by chevrons.

Character formats always come in sets – a start format flag and an end format flag. The start format flag establishes the character format. That character format is in force until the end format character flag is encountered, or until the document ends.

For example, a set of character format flags might look like this:

```
The book in question, «ULINE»Understanding
Radio Technology«END.ULINE», was first printed in 1952.
```

In this example, everything between the two flags — namely, the title of the book — will be underlined.

Remember to insert the end format flag at the point in the script where the character formatting is to end.

- ☛ The special flags `FONT1`, `FONT2`, and `FONT3` are used to create custom character format flags. See “Defining Printers for the Merge Processor” later in this chapter for details.

Field Flag Format Specifications (Alt-F4)

Anytime you include a field flag in the script, the data will appear in the script exactly as it exists in the record. For example, if data is all upper case in the record, it will appear this way in the final document.

An exception is a field defined in the dictionary with an output format. If a field definition includes an output format, this specification will be applied to the data when it is merged into the script.

You can include formatting specifications in the field flag itself if your script requires it. For example, if you wish to print a field using a specific field length or justification, you can include these specifications in the field flag itself. This capability is used to create formatted blocks such as tables, to limit or expand the size of a field in the script, or to override output formats defined in the dictionary.

- ☛ If you use the reserved field flags `MRG.DATE` and `MRG.TIME`, you must specify a format with these flags in order to format the data correctly.

When you want to include a field flag in a script, pressing the [Alt-F3] key displays a popup of all available fields. After you select a field from the popup, you will see another popup of field format specifications. If you wish to designate a special format for a field, choose one of the options from this popup. You can also display the popup of field format specifications by pressing [Alt-F4].

The format specification must be included in parentheses following the field name, but still within the braces. Some of the field format specifiers require an additional parameter. The parameter provides extra information required by that field format specification.

For example, the following field flag includes a format specification:

```
{MRG.DATE (OFV, D2 / ) }
```

In this example, the field format specification OFV is applied to the field flag MRG.DATE. The parameter "D2/" is used to indicate that the current date is to be output in the format MM/DD/YY.

The format specifications available are:

Format Specifier	Meaning
OFV	Specifies a justification, length or conversion, and a limit to the number of multivalues displayed.
NAMECAP	Converts text to first letter uppercase, other letters lowercase.
MV2COMMA	Converts multivalued fields into comma delimited lists.
COMP	Does comparisons of field values against fixed values.
TOUPPER	Converts text to all upper case.
TOLOWER	Converts text to all lower case.

Several of the field format specifications require an additional parameter. For example, the OFV specification requires a parameter that indicates the proper justification and length, conversion, or value limit for the field. These are explained below.

OFV The parameter for this format specifier consists of three parts. The parts are separated with ; (semicolon). They indicate an output format, a justification and length (format), and a limit on the number of values displayed.

For example, the following OFV format specifier might be used in a script:

```
{INVOICE.AMT (OFV, MD2$; R#10; 5) }
```

In the example, all three parts of the OFV parameter are included. The parameter MD2,\$ specifies that the invoice amount is to have two decimal places and a floating comma and dollar sign. The second part of the parameter indicates that the invoice amount will be right justified in a field 10 characters wide. Finally, the last part of the parameter dictates that only the first five values of the multivalued invoice amount field are to be output.

If one part of the parameter is to be skipped, its place is still held by ; (semicolon). For example, the following format specifier skips the first part of the parameter:

```
{ INVOICE . AMT ( OFV , ; R#10 ; 5 ) }
```

This example is like the previous one, but does not include a format specification for the output conversion.

For more information on the use of output formats, see “Data Formatting” in the “Adding Advanced Features to Windows with Paint” chapter in *Developing an Application*.

MV2COMMA

This format specifier turns a list of multivalues into a list separated by commas. The parameter added to this specifier can be any word such as “and” or “or” that will be used before the last word in the list.

For example, you might have a multivalued CONTACT field containing this data:

```
Mike
Joyce
Zachary
```

The following field flag includes the format specifier MV2COMMA and the parameter “and” for the contact name field above:

```
{ CONTACT ( MV2COMMA , and ) }
```

Using this example, the resulting output would be:

```
Mike , Joyce and Zachary
```

COMP

This format specifier is used primarily when creating conditional blocks. COMP requires a parameter that includes both the comparison operator (=, <, [, etc.) and a value to compare against the contents of the field. The COMP specifier returns a true or false value based on the comparison. That value is in turn used to determine whether the conditional block is printed.

- ☛ When a field is used as a condition, the COMP specifier returns a value of 1 (true) or 0 (false). Put ? (question mark) as the first character in the field flag to prevent this value from printing.

The following script fragment illustrates the use of the COMP field format specifier:

```
«IF»Since you are {?AGE (COMP,>59)} {AGE}
years old, don't forget that you are
eligible for our senior discount!«ENDIF»
```

In the example, the AGE field is used as a condition. When the contents of the field passes the test, the conditional block will print. A ? (question mark) is used as the first character in the field flag to designate that the AGE field is used as a condition, but that results of the comparison should not be printed. This prevents the COMP value (a 1) from printing. To print the actual age of the recipient, the AGE field flag must appear again in the conditional block.

The same comparison operators are used for COMP as are used for Query. For more information, see the "Using Query" chapter in *Accessing Data*.

You can use multiple format specifiers in one field flag. If you do, they are processed in order left to right. The output of the first field format specifier becomes the input for the next one.

For example, assume that you have a multivalued field containing the contact names ISABEL, MAX, and ROXANNE. The following example uses two field format specifiers to format this field:

```
{CONTACT (NAMECAP) (MV2COMMA, and) }
```

The end result is this text:

```
Isabel, Max and Roxanne
```

The field format specifier NAMECAP has converted the names into upper and lowercase names. The specifier MV2COMMA has turned the multiple values in this field (after they have been converted into lower case) into a string with commas and the word "and" between the values.

Testing a Script

While you are creating a script, you have the means to test your work. This includes a special Merge script debugging tool, as well as the opportunity to test run your merge.

The Merge Debugger

The Make Merge window menu, called with [F10] from any prompt in the window, has three script debugging options. There are separate options for debugging the scripts for the heading, footing, and the Merge script itself.

The script debug process reads through the appropriate script looking for errors. Errors include invalid field, format or character flags, unmatched sets of flags (for instance, «IF» with no «ENDIF»), or field format specifiers that are not defined.

When you debug a script, the debug process reads through and proofs the script. It then reports the number of errors it has found. To review the errors, enter the prompt for the script, and press [Alt-F7]. This presents a popup of all errors (in reverse order). If you choose an option from the popup, the cursor is placed on the error, and you can correct it. Errors that you have already chosen from the debug popup are flagged with an "X".

Testing the Merge Template (Alt-F8)

In addition to using the Merge Debugger, you can test your script by actually merging data with it. Two methods are available to do this.

The first is to press [Alt-F8] while editing the Merge script. This keystroke merges one data record with the current script and sends the output to the file entered at the *Output* prompt. The resulting document is displayed for your review.

If you did not enter a file name at the *Output* prompt, the merge output is sent to the VOC file using the data record key.

When you press [Alt-F8], you are prompted to enter the record key of a record to merge with the current template. Enter the key of any record in the data file. The Merge processor reads the data record, merges it with the template, writes the document to the output file, and displays it on the screen.

If you have already executed an [Alt-F8] review while editing this Merge template, you will see a prompt asking if you wish to review the existing test record. This enables you to see again the record you created earlier. If you enter N at this prompt, a new review document is created and displayed.

When you are done reviewing the sample record, press [Esc] to return to the Merge Script window.

A second method of testing the script is to execute a test run. This is an option available from the Make Merge menu. Press [F10] at any prompt to display the menu, and choose the **Testrun** option.

You will see the Merge Test window. There are two prompts: *Option* and *Number of Records*. The *Option* prompt allows you to specify that the output of the merge process is to go to the screen (terminal) or to a file, rather than to the printer. Press [F2] to see a list of options available at this prompt. These are the same options as those discussed under “Executing the Merge Process” below.

The *Number of Records* prompt is used to specify how many records from the data file are to be used in the Testrun. The Merge process will randomly pick this number of records and use them with the Merge template. When the test run is finished, you will be returned to the Make Merge window.

Executing a Merge

Once a Merge template has been created and tested, you are ready to execute a merge. The Merge process itself consists of executing a command that reads the Merge template, and then reads a record from the data file associated with the template. Data from the record is merged with the template, and the resulting document is sent to the screen, printer, or file. The process of reading and merging the record with the template is repeated until the list of records to be merged is finished.

Formatting Text and Resolving Flags

In addition to simply outputting the script text, the Merge process attends to several other tasks. The first of these is to format the paragraphs. Formatting consists of inserting any data required by the field flags, and then applying the paragraph formatting to the resulting text. If the insertion of data with a field flag has resulted in a long paragraph, for instance, the Merge process reformats the paragraph into as many lines as necessary, and even splits it across pages if required.

Field and format flags are processed right away. Character flags are handled a little differently. These are discussed below under “Merge Process Output”.

Merge Process Output

The default behavior of the Merge process is to send output to the printer. Even if no printer has been designated at the *Printer* prompt, output from the Merge process is routed to the current printer device for your computer.

You have options, however, to route the Merge output to either the screen (terminal) or to a file. If you send output to the screen, the document created for the printer is displayed on the screen. Paragraph and character formats are interpreted for the printer type designated in the Merge template. Unless you use the **X** (strip) option, this may result in output that does not display properly on the screen.

Output from the Merge process can also be sent to an Advanced Revelation file. If you choose this option, the formatted text of a document is sent to the file designated at the *Output* prompt in the Make Merge window. One record is created in the output file for each document merged with a data record. The records are written to the output file using the record key of the records in the data file.

Unlike those sent to the printer or screen, documents in the Output file are not ready to print. All page and paragraph formatting is complete, but character formatting such as underlining, bold, and italic lettering remains encoded. Page breaks remain encoded as well. For example, a character flag such as «ITAL» remains in this format when written to the Output file. A page break appears as the flag «FF».

To interpret the character flags, call the Merge process again with a special option (the **O** option). This option is used to convert these character formats into their respective printer codes while the document is printing or displaying.

Because character flags are not interpreted when output is written to a file, you can review the output without interference from control characters that may appear in the text. In addition, this permits you to create output that is printer independent. The file containing the output can be moved to another computer, and the Merge process on that copy of Advanced Revelation can interpret the character formats according to the printer available there.

When you call the Merge process to process records in an output file, you will need to specify the name of a template to process. The Merge process will use only two prompts from the template: *Output* and *Printer*. The other information — the script, default format, etc. — is already encoded in the output file. You may wish to create a special template called Output to provide these pieces of information.

Finally, a special option (the **X** option) creates output that is formatted, but from which all character flags have been stripped. This is useful for reviewing formatted text without seeing the character flags, or for printing to devices that accept no character formats (such as the screen).

- ☛ If you wish to create an operating system file ready to be sent to the printer (with all character formatting interpreted), use the PDISK command *before* you begin the Merge process. Use PDISK to route printer output to an operating system file, then use the Merge process with the P option to print to a printer. For more information see PDISK, in the “TCL Command Reference” chapter in *TCL*.

Creating Merge Output

The Merge process is available by selecting the **Access-Report-Merge** options from the Advanced Revelation Main menu. This displays the Merge window.

At the *Merge Name* prompt enter the name of a Merge template you created earlier using the Make Merge window. Press [F2] to see a list of available Merge templates.

The next prompt allows you to specify options for the Merge process. The options can be used to override temporarily the output specifications in the Merge template. In addition, three of the Merge options enable you to manipulate the character formats and page breaks assigned in the Merge script.

The options acceptable at this prompt are:

Option	Meaning
P	Send Merge output to the printer. Character flags are interpreted as printer codes.
T	Send Merge output to the screen (terminal). Character flags are interpreted as printer codes.
F	Send Merge output to the Output file assigned in the Merge template. Character flags and page breaks are encoded but not interpreted.
O	Send Output file text to the output device, turning character flags and page breaks into printer codes.
X	Send Merge output to the output device, stripping character flags and interpreting page breaks.

For a discussion of the differences between the file output options, see “Merge Process Output” above.

You can specify more than one option. For example, you might specify both the T and X options to display a document on the screen (terminal) while stripping character flags.

The Merge Processor can also be invoked from TCL. To do so, use this syntax:

```
MERGE template_name {key_list} {(option(s))}
```

Template_name is the name of a Merge template created with the Make Merge window. You can include a list of record keys in the command. These will be keys to records in the data file accessed by the Merge template. Finally, you can specify in parentheses the same options available in the Merge window.

Special Features of the Merge Processor

A variety of special features can help you customize the Merge processor to your own requirements. These include creating an additional script that can be inserted into the basic script, adding printer definitions to those already provided, importing scripts from outside of Advanced Revelation, and creating custom field format specifiers.

Special Text

Using the «SPECIAL» flag, you can have the Merge processor insert a script from another file into the current script. This enables you to create custom paragraphs, PS messages, etc., that can be merged into an otherwise generic script.

When the Merge processor encounters the «SPECIAL» flag, it looks into the file designated at the *Special Text* prompt in the Make Merge window. If a record is found in that file with the same record key as the key of the current data record, the script from the Special file is inserted at the flag. If there is no record in the Special file with the same key as that of the current data record, no insertion is done.

The script in the Special file follows exactly the same conventions as those for ordinary scripts. All formatting from the current script applies to the Special script.

To create the Special text, use the Advanced Revelation full-screen Editor. You will need to insert character, format and field flags by hand, since no popups are available in the Editor to provide these. If your Special text is complex, you can create it using the Merge Script window, and then use the Editor cut and paste buffers to move it to the Special text file.

Defining Printers for the Merge Processor

The Merge processor is provided with several printer definitions already defined. To see these, press [F2] at the *Printer* prompt in the Make Merge window.

You can create your own printer definition if none of the printers provided meets your requirements. To create a printer definition, you will need to know what escape or control character sequences your printer uses to begin and end character formatting such as underline, boldface, italic, etc. If you do not know these codes, you can find them in the documentation that accompanies your printer.

To create a new definition, select the **Printer** option from the Make Merge menu. This will display the Merge Printer Definition window. At the *Printer* prompt enter the name of the printer for which you are creating definitions. Do not use spaces in the name you specify.

The window contains prompts that enable you to define the start and stop codes for most kinds of character formats (bold, underline, etc.). In addition, the *Initialize* prompt enables you to specify a code that your printer uses to clear all previous settings. Finally, the *Font n* prompts enable you to define character fonts not already defined in the window.

To define any type of character format in this window, enter the ASCII value(s) of the characters or sequences used by your printer to achieve that format. Multiple ASCII values are separated with spaces.

For example, suppose that your printer manual specifies that the code for beginning to underline is the Escape character plus the sequence “p~H”. You would enter this string at the *Underline* prompt under the column labelled Begin:

```
27 112 126 72
```

Default Merge Template

You can create a default template using the Make Merge window. When you subsequently create new Merge templates, the values you have specified in the default template will be the default values for the new template.

To create a default template, enter the name DEFAULT at the *Merge Name* prompt. Enter values at all prompts for which you wish to establish defaults, including the scripts. Save your default template by pressing [F9].

Importing Scripts

You are not limited to creating scripts using the *Merge script* prompt in the Make Merge window. You can use any editor that can write out an ASCII text file.

The **Import** option from the Make Merge menu allows you to read an ASCII text file and import it into a Merge template as a script. To import a text file, press [F10] at any prompt to invoke the Make Merge menu.

Choose the **Import** option. When you do, you will be prompted to enter the name of an operating system file. Enter the name, including the full path name.

You will then be asked if you wish to wrap long lines. If you accept the default Y at this prompt, the Merge import process will format long lines to fit into the Merge Script window (80 columns wide). If you enter N at this prompt, long lines will run off the edge of the screen (but will be formatted properly when you merge the document).

Creating Custom Field Format Specifiers

If you have field formatting requirements not addressed by the field format specifiers provided in the Merge processor (see “Field Flag Format Specifications” above), you can create your own field format specifiers. Custom field format specifiers are created as R/BASIC subroutines.

When the Merge process calls a field format specifier, it loads the system variable @ANS with the data for that field. If a parameter is required for the field format specifier, this is passed as an argument. When the format specifier has formatted the data, it reloads @ANS with the formatted version of the data.

For example, you might create a custom field format specifier called FMT. To call the custom routine from a script, use it as you would any other field format specifier. For example, this field flag contains the format specifier FMT:

```
{ INVOICE . DATE ( FMT , D2 / ) }
```

In this example, the FMT specifier takes one parameter.

FMT will be an R/BASIC subroutine that has been cataloged. An example of the code for FMT might be:

```
SUBROUTINE FMT( FORMAT )
FIELD.DATA = @ANS
IF FIELD.DATA NE '' THEN
    FIELD.DATA = OCONV(FIELD.DATA, FORMAT)
    * check that conversion was successful
    IF STATUS() EQ '' THEN NULL ELSE
        @ANS = FIELD.DATA
    END
END
RETURN
```

This example accepts field data in @ANS, and converts it using the R/BASIC OCONV function. The type of conversion to be applied to the data is passed as the parameter for the field format specifier. The reformatted data is passed back to the Merge processor in @ANS.

Table H6-2

Quick Reference Chart for the Merge Processor

Keystrokes	Meaning
[F2]	(at a script prompt) Enter script edit window
[F2]	(in a script edit window) Display popup of flag popups
[Alt-F1]	Format flag popup
[Alt-F2]	Character flag popup
[Alt-F3]	Field flag popup
[Alt-F4]	Field format specifiers
[Alt-F5]	Edit field format specifiers popup
[Alt-F7]	Script debug popup
[Alt-F8]	Perform merge sample
[Alt-F9]	Begin format/character flag («)
[Alt-F10]	End format/character flag (»)
[F10]	(at a Make Merge prompt) Display Make Merge menu
	<i>(continued)</i>

Quick Reference Chart for the Merge Processor (Continued)

Keystrokes	Field Flags	Meaning
{ <i>fieldname</i> }		Insert data from the field <i>fieldname</i>
{MRG.PAGE}		Insert current page number
{MRG.TIMEDATE}		Insert current time and date
{MRG.DATE}		Insert current date (internal format)
{MRG.TIME}		Insert current time (internal format)
Keystrokes	Format Flags	Meaning
[Alt-F]	«FORMAT»	Establish default format for paragraph
[Alt-U]	«UNFORMAT»	Disable default format for paragraph
[Alt-I]	«IF»	Begin conditional text block
[Alt-L]	«ELSE»	Begin alternate text block
[Alt-E]	«ENDIF»	End conditional or alternate block
[Alt-P]	«PAGE»	Force new page
[Alt-S]	«SPECIAL»	Insert special text
[Alt-V]	«VBLOCK»	Begin multivalued block
[Alt-K]	«END.BLOCK»	End multivalued block
[Alt-H]	«VBLOCK.HEAD»	Define start of value block heading
[Alt-C]	«CENTER»	Center line of unformatted paragraph
[Alt-N]	«ENDCENTER»	End centering of lines
[Alt-R]	«RIGHT»	Right justify remainder of unformatted line

(continued)

Quick Reference Chart for the Merge Processor *(Continued)*

Keystrokes	Format Flags	Meaning
[Alt-J]	«PARA»	Force paragraph without space
[Alt-A]	«PARA.INDENT, <i>n</i> »	Establish paragraph indent of <i>n</i> characters
[Alt-W]	«DBLSPC»	Double space lines

Keystrokes	Character Flags	Meaning
[Alt-Z][B]	«BOLD»	Start bold character
[Alt-Z][E B]	«END.BOLD»	End bold character
[Alt-Z][U]	«ULINE»	Start underline character
[Alt-Z][E U]	«END.ULINE»	End underline character
[Alt-Z][M]	«EMPH»	Start emphasized character
[Alt-Z][E M]	«END.EMPH»	End emphasized character
[Alt-Z][D]	«DBL»	Start double-wide character
[Alt-Z][E D]	«END.DBL»	End double-wide character
[Alt-Z][S]	«SUPER»	Start superscript character
[Alt-Z][E S]	«END.SUPER»	End superscript character
[Alt-Z][L]	«SUB»	Start subscript character
[Alt-Z][E L]	«END.SUB»	End subscript character
[Alt-Z][I]	«ITAL»	Start italic character
[Alt-Z][E I]	«END.ITAL»	End italic character
	«FONT1»	Begin font1
	«END.FONT1»	End font1
	«FONT2»	Begin font2
	«END.FONT2»	End font2
	<i>(continued)</i>	

Quick Reference Chart for the Merge Processor *(Continued)*

Keystrokes	Character Flags	Meaning
	«FONT3»	Begin font3
	«END.FONT3»	End font3
Keystrokes	Field Format Specifiers	Meaning
	OFV	Specify conversion, justification/length, value limit
	NAMECAP	Convert text to first letter uppercase, other letters lowercase
	MV2COMMA	Convert multivalued fields to comma-delimited lists
	COMP	Compare field values against fixed values
	TOUPPER	Convert text to all upper case
	TOLOWER	Convert text to all lower case

1

2

3

Chapter K 7**Programming with SQL**

Introduction	K 7.1
The SQL Module Language	K 7.1
Embedded SQL	K 7.2

Chapter K 8**SQL Reference**

Extensions to SQL	K 8.1
Reserved Words	K 8.4
Advanced Revelation SQL Data Types	K 8.5
SQL Operators and Notation	K 8.6
SQL Keywords	K 8.7
SQL Statements	K 8.30

Chapter L 1**Introduction to Networks**

Introduction	L 1.1
Advanced Revelation and Networks	L 1.3

Chapter L 2**Managing Advanced Revelation on a Network**

Managing Files on a Network	L 2.1
The Advanced Revelation Environment	L 2.2
Executing Network Commands	L 2.5
Managing Indexing on a Network	L 2.6
Other Network Management Issues	L 2.8

Chapter L 3**Locking**

Introduction to Locking	L 3.1
Types of Locking	L 3.2
Establishing Locking	L 3.5
Setting Locking Options	L 3.7
Implicit Locking	L 3.8
Additional Lock Issues	L 3.13

Chapter M 1**Introduction to Environmental Bonding**

Environmental Bonding	M 1.1
Tenets of Environmental Bonding	M 1.3
Using an Environmental Bond	M 1.4
Characteristics of Environmental Bonding	M 1.8

Chapter M 2**The ASCII Environmental Bond**

Description of the ASCII Bond	M 2.1
Using the ASCII Bond	M 2.3

Chapter M 3**The dBASE III Environmental Bond**

Description of the dBASE III Bond	M 3.1
Using the dBASE III Bond	M 3.3
The dBASE III Bond Reference	M 3.12

Chapter M 4**The SQL Server Environmental Bond (Optional Module)****Volume III****Chapter N 1****Introduction to R/BASIC**

R/BASIC Overview	N 1.1
----------------------------	-------

Chapter N 2**Features of R/BASIC**

Features of R/BASIC	N 2.1
-------------------------------	-------

Chapter N 3**Programming with R/BASIC**

Programming with R/BASIC	N 3.1
------------------------------------	-------

Chapter N 4**R/BASIC Command Reference**

R/BASIC Command Reference	N 4.1
-------------------------------------	-------

Chapter N 5**R/BASIC System Subroutines**

R/BASIC System Subroutines	N 5.1
--------------------------------------	-------

Chapter I 1

Introduction to TCL

TCL (The Command Level) is Advanced Revelation's interactive command language. There are TCL commands for most Advanced Revelation functions, such as accessing windows, creating files, running reports, and more.

This section provides an overview of TCL. The first two chapters introduce TCL, and summarize the commands that you can execute at TCL. The third chapter is a detailed reference for all TCL commands, including descriptions, syntax, and examples.

Introducing TCL I 1 . 1



Introducing TCL

TCL (The Command Level) is Advanced Revelation's interactive command language. It is the "coordinator" in Advanced Revelation that ultimately controls the execution of menus, windows, utilities, R/BASIC programs, and other processes. As the monitor program, TCL prompts for commands, executes them, and returns to await further user input.

Users will find in TCL a close analog to an operating system command line. In both cases, a master program presents the user with a prompt, accepts user input, processes the command, and then returns with another prompt awaiting the next command. In your operating system, the ready state is usually indicated by the familiar A or C prompt; in TCL, the prompt is a colon (:).

From TCL, you can directly execute any of dozens of commands. These commands range in function from maintaining files and records, to generating custom reports, to creating and executing R/BASIC programs, to invoking Advanced Revelation's window-driven application processor. The list of commands that TCL will accept is limited only by what is found in a user's vocabulary (VOC) file.

Advanced Revelation's menu-driven application processor serves as a "shell" or "front end" for TCL. Most of the functions accessible at TCL are available as window-driven options from Advanced Revelation's menu system. In fact, the menu and window system in most cases is simply executing a TCL-level command, but is lifting from the user the burden of knowing the syntax, spelling and options of the various commands.

Why Use TCL?

Given that most functions can be accomplished from a window, why is an understanding of TCL desirable? There are several reasons.

First, direct execution of TCL commands is faster than using a prompt window to collect and then process the information. Experienced Advanced Revelation users find that the direct control TCL provides is frequently preferable to the indirect access of a window or menu.

Second, some TCL commands may not be available through menus or windows. Advanced Revelation allows you to edit or append the list of acceptable TCL commands at any time by making new entries in the vocabulary (VOC) file. TCL will make all of these commands available where a menu might not.

There are other reasons as well. TCL makes available a batch-oriented mode, so that you can stack up a series of commands to be executed in sequence, much like a native batch file. This batch processing would be difficult or impossible if all commands were driven from interactive menus and windows.

Finally, TCL commands can be embedded into lower level R/BASIC programs. Using this powerful capability of R/BASIC requires a good knowledge of how to use TCL.

Accessing TCL

TCL can be accessed in one of two ways. The most direct method is to use the [F5] key. The [F5] key will bring up the TCL window from anywhere except the debugger. When TCL processing is finished, pressing [Esc] at the TCL window will return control to any process that was suspended.

Alternatively, you can choose the TCL option from the Main-Exit menu sequence. Pressing [Esc] at this TCL level returns you to the Exit menu.

The TCL Window

The TCL window is active whenever TCL has been invoked. The colon prompt in the left margin indicates that TCL is ready to accept commands and process them.

Commands are typed in as a single line at the prompt. When you type a command, pressing [Enter] or [F9] (Save) will cause TCL to begin processing.

Once the command is completed, the TCL window will reappear, and you can proceed with more commands or can exit the window.

Valid Keys

Because TCL is much like other windows, standard Advanced Revelation key definitions are applicable. These key definitions are discussed in more detail in *Getting Started*.

One difference in TCL is that the options key [F2] will produce a popup window of previously executed commands. (The exact number of commands that will be available is dependent on the setting established in the environment window.) These commands can be re-executed, saving the time and trouble of having to re-key a command. These commands can also be saved to file and given a name, so that they may be recalled later with a single-word command. See "Saving TCL Commands" below.

Softkeys

The [F6] key will display a list of the softkeys that are active in the TCL window. The softkeys are defined as follows:

- [Shift-F1] This key brings up a window that allows you to link multiple TCL commands or store one or more commands in the VOC file under a single name. For more details about this process, see "Linking Multiple Commands" below.

- [Shift-F2] This key produces a popup window listing all the commands that can be executed at TCL. The listing includes any synonyms for the commands, as well as the name of the menu on which they appear. You can choose which command to execute by using the cursor keys to highlight a command and pressing [Enter].
- [Shift-F3] This key produces an expression window, which enables you to define a symbolic field "on-the-fly" while querying a file. With this expression window you can create fields as they are needed without exiting the current query session. The fields created with this window are stored in the dictionary of the file being queried.
- [Shift-F4] This key produces a list of all Advanced Revelation files that are currently attached.
- [Shift-F5] This key produces a list of all volumes that are currently accessible.

Linking Multiple Commands

Entry of single commands is accomplished by simply typing the command and pressing [Enter] or [F9]. It is possible, however, to enter a series of commands to be executed in sequence. This enables you to "stack" commands to be executed automatically.

For example, a report can be created by stacking a series of commands such as these:

```
SELECT CUSTOMERS BY ZIP
SAVELIST CUST.LIST
GETLIST CUST.LIST
LIST CUSTOMERS NAME ADDRESS CITY ST ZIP (P)
```

In this example, Advanced Revelation will sort the CUSTOMERS file by ZIP code, save a list of record keys representing the sorted file, get the list again, and then run an R/LIST report. If this set of commands were not executed in series, an operator would need to attend the machine to enter a new command as each previous one finished.

You can enter a series of commands using the Multi-Line Command Window. This window is accessed by pressing [Shift-F1] from the TCL window.

The first prompt in the window is the multiline prompt *Commands*. Enter a series of TCL commands one after another, separating each by pressing [Enter] after each one. When all commands are entered, pressing [F9] will start execution of the series. When the last command is finished, the TCL window will once again appear.

Saving TCL Commands

Any TCL command can be saved under a user-designated name in the VOC file. This enables you to create long or complex commands and give them an easy-to-remember name, or to create synonyms for existing TCL commands.

For example, the series of commands illustrated in the section above might be saved under the name CUSTOMER.REPORT. Once this command is saved, a user can simply type in the words CUSTOMER.REPORT rather than having to re-enter the full series of commands each time.

There are two ways to save TCL commands in the VOC file. One is to use the Linking Multiple Commands window, and the other is to use Advanced Revelation's Editor to create a special vocabulary record for your commands.

To use the Linking Multiple Commands window, bring it up from the TCL window by pressing [Shift-F1]. At the *Commands* prompt, enter the series of commands to be saved as described above under "Linking Multiple Commands." Rather than pressing [F9] to execute the commands, however, move to the next prompt, *Save Record Id*, by pressing [Enter] after the last command has been entered.

At the *Save Record Id* prompt, enter in a name under which you wish to recall the command(s) later. Use a name that is not already defined in the VOC file. Do not use spaces or quote marks in the name.

If you wish both to save the command and immediately execute it, move to the next prompt, *Save Only*, and enter N. If you wish only to save the command and not execute it, leave this field with Y. Use [F9] to save the new TCL command.

You can also use Advanced Revelation's editor to create new VOC entries. Use the EDIT command to put an entry into the VOC file. The record key should be the name under which you wish to recall the commands later.

The first line of the VOC entry must contain the word "TCL", so that TCL will later recognize this as a stack of TCL commands. Each subsequent line is a new TCL command. A record in the VOC file to execute the command stack illustrated above would look like this:

```
TCL
SELECT CUSTOMERS BY ZIP
SAVELIST CUST.LIST
GETLIST CUST.LIST
LIST CUSTOMERS NAME ADDRESS CITY ST ZIP (P
```

Batch Processing of TCL Commands

TCL provides special keywords and symbols for use when a stack of TCL commands has been saved to be executed later.

These keywords and symbols are used primarily for batch processing:

- An asterisk inserts comments or remarks into a VOC item.
- ! An exclamation point designates that the text following is to be output when that line of the saved command is executed.
- DATA** Stacks up words in the input buffer. These words are retrieved by subsequent prompts in the next process(es).
- ? A question mark causes TCL to prompt for information as the batch process is being run. The ? operator can be used anywhere in a TCL command to prompt for a word or phrase appropriate at that point of the command. It includes the facility of creating a prompt message.

These are discussed in more detail later in this section.

Interfacing with Command Windows

Although TCL commands are fully independent of the applications processor of Advanced Revelation, they can still make use of existing windows.

Many commands that have an associated window can call that window. To do this, enter the command with no additional parameters. For example, to call the window for the EDIT command, enter the word EDIT with no file name or record key list. When TCL detects that the command is not complete, it will call the appropriate window and prompt for the missing information.

Certain windows in the application processor do not have corresponding TCL commands. For example, the Environment window does not have a TCL correlative. The window can be executed directly from TCL, however, by making a call to the appropriate window processor. If the window to be called is an Advanced Revelation command window, the window processor is CW. If the window is an applications window, the window processor name is AW.

For example, to call the Recordcopy window from TCL, type this at TCL:

```
CW RECORDCOPY
```

or to call an applications window called CUSTOMER.ENTRY, type this:

```
AW CUSTOMER.ENTRY
```

When you exit the window, the TCL window will once again appear.

Using TCL from R/BASIC

TCL commands can be called directly from within an R/BASIC program. This permits a developer to use existing Advanced Revelation utilities and commands in a program without having to re-write the logic to accomplish these tasks.

For example, a developer may wish to create a new file from within a program. Rather than writing special processes to do this, the developer can call the TCL MAKEFILE command from within the program.

The R/BASIC commands PERFORM and EXECUTE are used to interface to TCL from within a program. For details on using these commands, refer to *R/BASIC*.

Chapter I 2

Features of TCL

This chapter provides a summary of all TCL commands, arranged by function. Review the listing of commands if you have a specific function in mind but do not remember what TCL command is used to execute it.

Features of TCL I 2 . 1



Features of TCL

You can perform any operation at TCL that you can perform using the Advanced Revelation menus. The TCL commands are grouped by category and briefly described below.

Account Access

These commands enable users to make and access new Advanced Revelation accounts.

- MAKEACCOUNT** Creates a new Advanced Revelation account.
- DELETEACCOUNT** Removes an existing account from the system.
- LOGTO** Logs from the current Advanced Revelation account to another existing account.
- PASSWORD** Adds or modifies the password for the current account.
- Batch** Batch commands can be embedded into "batch" items in the VOC file in order to permit control over processing.

Environment

Environment commands perform system functions and establish the Advanced Revelation environment.

- FLUSH** Forces Advanced Revelation to write any new data in RAM to disk.
- LOGOFF** Leaves Advanced Revelation and returns control to the operating system.
- MACRO** Activates an existing macro key set.
- PC** Suspends Advanced Revelation and exits temporarily to the operating system. Type EXIT at the operating system command line to return to Advanced Revelation.
- PDISK** Directs printer output to a file or to another printer.
- RESET** Reinitializes the Advanced Revelation system.
- RETURN** Returns from one recursive execution level to the next highest level.

ROLLOUT	Specifies an operating system file for storing the Advanced Revelation status. A SUSPEND command will write to this file.
SET-CRT	Sets the width and height of the terminal screen.
SET-LPTR	Sets the width and height of the printer.
SET-OPTIONS	Sets the default system options (printer output, disk write, etc.).
SUSPEND	Stores the current state of Advanced Revelation to a file and reloads a copy of the operating system. Type EXIT at the operating system command line to return to Advanced Revelation. Unlike the PC command, Advanced Revelation is "rolled out" of memory to free up RAM.
TIME	Displays the current time and date.
WHO	Displays the current user name, account name, system environment, network environment, and installed options.

File Access and Information

File access commands provide the means for controlling and viewing the distribution of files on multiple drives, directories, or volumes.

ATTACH	Makes files available from a drive or a volume other than the default volume.
DETACH	Removes files on a given drive or volume from the list of currently available files.
LISTFILES	Lists all available attached files for the current account.
LISTINDEX	Lists index information for indexed files.
LISTVOL	Lists all files available from a volume, whether attached or not.
LISTVOLUMES	Lists all attached Advanced Revelation volumes for the current account.
LOCKS	Displays information about current locks.
NAMEVOLUME	Initializes a volume directory for files

File Maintenance

File maintenance commands enable users to create, delete, and maintain files.

BUILD	Runs a batch update process defined at the Batch window.
CLEARFILE	Clears the contents of the data or dictionary portion of a file without deleting the file.
COMMITLOG	Displays or purges a commit log file.
CONTROL_OFF	Removes transaction and domain validation controls from a file. The controls must have been previously installed with the CONTROL_ON command.
CONTROL_ON	Installs transaction and domain validation controls for a file.
DEFINEFILE	Creates a new file and defines the file structure.
DELETEFILE	Deletes a file or removes bonding.
DICT	Displays the Dict window for editing field definitions.
DUMP	Examines the contents of each frame of an Advanced Revelation file.
ENCRYPT	Encrypts a window or form template.
EXPORT	Copies an Advanced Revelation file to a foreign file format.
FILECOPY	Copies entire files to new files or volumes.
IMPORT	Copies a foreign file to Advanced Revelation file format.
INDEX	Updates the existing index for a single field in a file.
LISTDICT	Displays the field definitions of a file's dictionary.
REDEFINEFILE	Restructures a file.
RENAMEFILE	Renames a file.
SETFILE	Creates a synonymous name for an existing file.
SETVOLUME	Creates a volume identifier.
TRANSACTION	Controls transaction processing.

Filtering

Filtering commands are used to create and maintain selected subsets of data. In all cases the activated list refers to a list of record keys meeting the criteria used to find the records. The command following the activation of a list will typically use only the set of records represented by the list.

CLEARLIST	Deactivates a currently selected list.
DELLIST	Deletes a list previously saved with SAVELIST.
FILTER	Runs an existing filter record to create a select list.
GETLIST	Retrieves a list previously saved with SAVELIST and makes those records available for further processing.
MAKELIST	Creates a select list directly out of data on file.
QUERY	Displays a data entry window in Query mode. Query mode is used to create a select list.
SAVELIST	Saves the results of a SELECT statement.
SELECT	Selects items from a file for further processing.
SSELECT	Selects items from a file for further processing after sorting them by record key.

R/BASIC

R/BASIC commands are used to create, compile, and list R/BASIC programs.

BLIST	Prints a formatted listing of R/BASIC source code to the screen or printer.
COMPILE	Compiles R/BASIC source code into R/BASIC object code.
EDIT	Creates or modifies R/BASIC programs.
EVAL	Compiles and executes R/BASIC source code.
MAKEVOC	Catalogs an R/BASIC program. This puts a program pointer for the program into the VOC file.
RUN	Executes an R/BASIC program.
UNMAKEVOC	Removes an R/BASIC program entry from the VOC file.

Record Maintenance

Data transfer/maintenance commands control the copying of data.

- | | |
|-------------------|--|
| DELETEREC | Deletes records in files. |
| EDIT | Creates or modifies records in files. |
| RECORDCOPY | Copies records from a file to other records, files, or to the screen or printer. |

Reports

When producing reports, the user has options regarding filtering, the display of data, headings and footings, and whether the report will be printed to the screen or printer.

- | | |
|--------------|--|
| COUNT | Counts the number of records in a file meeting specified criteria. |
| EW | Invokes the EasyWriter utility for creating reports. |
| FORM | Creates a report based on a template created at the Form window. |
| LABEL | Executes a report based on a label defined at the Make Label window. |
| LIST | Lists the contents of a file. |
| MERGE | Merges records with an existing Merge template (defined at the Make Merge window). |
| RLIST | Executes an R/LIST report created earlier using the Rlist window. |
| SORT | Lists the contents of a file, sorted by record key. |
| SUM | Sums the contents of one field of every record meeting specified criteria in a file. |

Terminal Emulation

Terminal emulation commands enable Advanced Revelation to enter into the terminal emulation mode suitable for communication with other computers.

- PORT** Displays the current communication parameters.
- TERMINAL** Enters terminal emulation mode and sets the configuration of the asynchronous port.

User Maintenance

User maintenance commands provide for the establishment and maintenance of user names within Advanced Revelation.

- BUMP** Enables four new users to be added to the networked system, and allows changes to the current network driver.
- DELETEUSER** Removes an existing user from the system.
- MAKEUSER** Adds a new user to the system.

Window and Menu Access

These commands enable you to access menus and windows from TCL.

- AW** Displays an application window.
- CW** Displays a command window.
- PAINT** Displays the Paint window to create, modify, and delete application windows and forms.
- PAINT.FORM** Displays the Paint window for forms creation.
- RUNMENU** Displays a menu.
- SQL** Displays the SQL window, or executes the specified SQL statement.
- TUTORIAL** Activates the Advanced Revelation Tutorial.
- WINDOW** Displays an application or command window.

Chapter 13

TCL Command Reference

This chapter comprises detailed information on using TCL commands. If you cannot remember the name of a particular TCL command, refer to the previous chapter, “Features of TCL”, for a summary of TCL commands arranged by function.

ATTACH	13.1
AW	13.2
Batch Commands	13.2
BLIST	13.4
BUILD	13.6
BUMP	13.7
CLEARFILE	13.7
CLEARLIST	13.8
COMMITLOG	13.9
COMPILE	13.9
CONTROL_OFF	13.11
CONTROL_ON	13.12
COUNT	13.13
CW	13.15
DEFINEFILE	13.15
DELETEACCOUNTT	13.17
DELETEFILE	13.18
DELETEREC	13.19
DELETEUSER	13.20
DELLIST	13.21
DETACH	13.22

DICT	I 3 . 23
DUMP	I 3 . 23
EDIT	I 3 . 25
ENCRYPT	I 3 . 25
EVAL	I 3 . 27
EW	I 3 . 27
EXPORT	I 3 . 28
FILECOPY	I 3 . 29
FILTER	I 3 . 30
FLUSH	I 3 . 31
FORM	I 3 . 31
GETLIST	I 3 . 32
IMPORT	I 3 . 33
INDEX	I 3 . 33
LABEL	I 3 . 34
LIST	I 3 . 35
LISTDICT	I 3 . 38
LISTFILES	I 3 . 39
LISTINDEX	I 3 . 40
LISTVOL	I 3 . 41
LISTVOLUMES	I 3 . 42
LOCKS	I 3 . 42
LOGOFF	I 3 . 43
LOGTO	I 3 . 43
MACRO	I 3 . 44
MAKEACCOUNT	I 3 . 45

MAKELIST	I 3 . 46
MAKEUSER	I 3 . 47
MAKEVOC	I 3 . 48
MERGE	I 3 . 49
NAMEVOLUME	I 3 . 50
PAINT, PAINT.FORM	I 3 . 51
PASSWORD	I 3 . 52
PC	I 3 . 53
PDISK	I 3 . 54
PORT	I 3 . 55
QUERY	I 3 . 56
RECORDCOPY	I 3 . 56
REDEFINEFILE	I 3 . 59
RENAMEFILE	I 3 . 61
RESET	I 3 . 62
RETURN	I 3 . 63
RLIST	I 3 . 64
ROLLOUT	I 3 . 65
RUN	I 3 . 65
RUNMENU	I 3 . 66
SAVELIST	I 3 . 67
SELECT	I 3 . 68
SET-CRT	I 3 . 70
SET-LPTR	I 3 . 71
SET-OPTIONS	I 3 . 72
SETFILE	I 3 . 72

SETVOLUME	I 3 . 74
SORT	I 3 . 75
SQL	I 3 . 78
SSELECT	I 3 . 78
SUM	I 3 . 80
SUSPEND	I 3 . 82
TERMINAL	I 3 . 84
TIME	I 3 . 86
TRANSACTION	I 3 . 87
TUTORIAL	I 3 . 88
UNMAKEVOC	I 3 . 88
WHO	I 3 . 89
WINDOW	I 3 . 90

ATTACH

The ATTACH command makes files on a specified or default volume available to the system. You can attach entire volumes or specific files on a volume.

```
ATTACH vol {DICT} {DATA} {file_list} {option}
```

Using Attach

Use ATTACH to access Advanced Revelation files on a volume other than the default drive (the default drive is always attached automatically). ATTACH is used to permit you to distribute data into more than one volume, including any combination of local, network, and logical drives or volumes.

ATTACH first reads the volume directory of the indicated volume to search for files belonging to the current account. It then searches the QFILES file to see if any Qfile pointers can be attached. If no QFILES file is attached, ATTACH searches the VOC file for Qfile pointers.

The names of all the attached files and Qfile pointers are maintained in the FILES file.

vol

Vol (volume) defines the volume for the files to be attached. Use a colon as part of the drive specification.

DICT

Use DICT (before a file name in *file_list*) to attach only the dictionary of a file.

DATA

Use DATA (before a file name in *file_list*) to attach only the data file.

file_list

Use *file_list* to specify the files to be attached.

In cases where two identically named files are on separate volumes, the system will use only the last volume attached.

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of ATTACH Command

```
ATTACH A: CUSTOMERS
```

The file CUSTOMERS on drive A: is attached.

ATTACH PM PD

Files PM and PD on the default volume will be added to the FILES file.

AW

The AW command displays the specified application window.

AW window

window

The name of an application window having a template in the TEMPLATES file.

Correct Use of the AW Command

AW CUSTOMERS

The application window named CUSTOMERS is displayed.

Batch Commands

Batch commands can be embedded into TCL batch processes. They include commands to embed comments, display text, prompt for input, and store user input.

For more information about creating and using TCL batch commands, see the “Introduction to TCL” chapter.

*** comment**

! output_text {options}

? prompt_message ?

DATA [response ::]

Any text that follows an asterisk will be ignored as the batch command executes. This permits you to insert comments or remarks for future reference.

!

Any text that follows an exclamation point will be output to the screen or printer as the batch command executes.

If you wish to have the text appear on more than one line, put | (vertical bar) between lines of the text.

options The options provide additional information about the output. They should be enclosed in parentheses. The options available for the ! command are:

Option	Meaning
(P)	Print the text to the printer
(U)	Display the text, and leave it. The text will disappear when a message with the D option is displayed, or when the batch process is finished.
(D)	Remove a message previously displayed with the U option.
(M)	Add to the message already displayed. You can also move it to a new location.

The options can include any one of the choices above, plus two additional parameters. The two extra parameters permit you to place the ! output at any column and row coordinate on the screen. For example, the following ! command displays a message at column 5, row 10:

```
!Selecting File (U,5,10)
```

Option D does not print text. It is used to remove a type U message and to provide a time delay for U type messages.

When using option D for a time delay, provide a number in place of the text. The message created with a type U message will display for that number of seconds.

? (question mark)

Question marks are used in pairs around the portion of a TCL command that is to appear as a prompt. You can substitute a prompt for any portion of any TCL command. Any text that appears inside the pair of question marks will be treated as a prompt message. If quotes or other special characters are required in the data, be sure that these are inside of the question marks.

DATA

Any text that follows the word DATA will be put into the keyboard buffer, to be extracted again at the next input prompt. Multiple responses can be stacked up by separating them with commas.

Correct Use of Batch Commands

```
!Welcome to Advanced Revelation (U)
!8 (D)
!Attaching Files|Please stand by (U)
ATTACH DATA
!Preparing Application (M,10,12)
RUN BP PREPARE.APPLICATION
!(D)
```

The first two lines cause the message "Welcome to Advanced Revelation" to be displayed for eight seconds. The message following displays two lines. While this message is displayed, the ATTACH command is executed. The next message adds to the existing one and moves the display to column 10, row 12. Finally, the last line of the record clears the message.

- ☛ If by coincidence the text you are displaying is the key to a record in the MESSAGES file, that message will be displayed instead of yours. To correct this problem, put a space in front of your message. For example:

```
!101
  Displays message number 101
```

```
! 101
  Displays the literal text "101"
```

* The following command selects active customers

```
SELECT CUSTOMERS WITH STATUS = 'A'
```

```
! Selecting records, please stand by ...
SELECT INVOICES WITH INV_DATE = TODAY
```

```
SELECT CUSTOMERS WITH ST = ?"Enter the state to select"?
```

```
DATA 01-01-87, 100
RUN BP POST.INVOICES
```

BLIST

BLIST formats and lists R/BASIC programs to the screen or printer.

BLIST filename keys {options}

Using BLIST

Use BLIST to display R/BASIC source code listings in a format that makes program listings easy to read.

BLIST will use the following format rules:

- Line numbers appear in the left margin.
- Loops and conditions are indented 3 spaces and are marked with an indentation mark (I).
- Labels are printed to the left of the code displayed.
- Lines too long for the output device are split and continued on the next line.

filename

Filename is the name of the file that contains the R/BASIC programs to be listed.

keys

Keys is a list made up of one or more record keys (program names) separated by spaces. Substitute an * (asterisk) as the record key to list all the records in a file. To perform BLIST using an active list of record keys, *keys* list must be empty.

options

Options must be enclosed within parentheses. More than one option can be used within a single set of parentheses. Five options are available.

Option	Meaning
(E)	Use E to suppress top-of-form before printing.
(I)	Use I to insert programs defined by \$INSERT statements.
(L)	Use L to suppress indentation marks (I).
(P)	Use P to output to a printer (default).
(T)	Use T to output to a terminal instead of a printer.

Correct Use of BLIST Command

```
BLIST BP TEST2
```

```
PROGRAM LISTING FOR TEST2
18:01:30 29 APR 1987
0001      EQU CLEAR.SCREEN   TO @(-1)
0002      EQU CLEAR.LINE     TO @(-4)
0003      EQU WITHOUT.PROMPT TO -1
0004      EQU NO              TO 0
0005      EQU OTHERWISE      TO 1
0006      EQU YES            TO 1
0007      EQU ESC            TO CHAR(27)
0008      * PROGRAM TO DISPLAY SEQ NUMBERS FOR OPERABLE KEYS
0009
*****
0010      PRINT CLEAR.SCREEN:
```

```

0011         DONE = NO
0012         LOOP WHILE NOT(DONE)
0013         |   REPLY = ''
0014         |   LOOP WHILE REPLY = ''
0015         |   | INPUT REPLY,-1
0016         REPEAT
0017         BEGIN CASE
0018         CASE REPLY = ESC
0019         | DONE = YES
0020         CASE LEN(REPLY) 1
0021         | PRINT @(0,1):CLEAR.LINE: "1ST SEQ=": ⇨
          SEQ(REPLY[1,1]):" 2nd SEQ = ":SEQ(REPLY[2,1]):
0022         CASE OTHERWISE
0023         | PRINT @(0,1):CLEAR.LINE: "SEQ = ":SEQ(REPLY)
0024         END CASE
0025         REPEAT
0026         STOP

```

BLIST formats the display to indent loops and conditions.

BUILD

BUILD runs a batch update process defined at the Batch window.

BUILD *record* {filename}

Using BUILD

BUILD executes a predefined batch update process, writing (building) a transaction record to the transaction file specified at the Batch window. You can access the Batch window through the **Tools-Records-Batch Update** options from the Main menu.

record

The name of the update process record defined at the Batch window.

filename

The name of the file where *record* is stored (defaults to the UPDATE file).

Correct Use of BUILD Command

```
BUILD TRANS.RECORD
```

The record TRANS.RECORD defines the batch update process that is executed, and must exist in the UPDATE file.

BUMP

The BUMP command displays the Bump Process window. The Bump Process window allows you to change network drivers or increase the number of users, when used with the Network Bump Disk.

- ☞ For networked versions of Advanced Revelation, see your Network Bump Disk documentation.

```
BUMP
```

Correct Use of BUMP Command

BUMP

Activates the Bump Process window

CLEARFILE

CLEARFILE deletes all information from the dictionary or data section of a file.

CLEARFILE is a synonym for CLEAR-FILE.

```
CLEARFILE {DICT} {DATA} filename {options}
```

Using CLEARFILE

CLEARFILE permanently removes all records from the data or dictionary portions of a file, but leaves the file on the system. Removing the records frees up disk space.

CLEARFILE will ask you to type in “Y” in order to confirm that you wish to delete all records in the file.

- ☞ CLEARFILE will *not* make use of an active filter. To clear only selected records, use the DELETEREC command.

DICT

Use DICT to delete all records from the dictionary of the file represented by *filename*.

DATA

Use DATA to delete all records from the data section of the file represented by *filename*.

- ☞ If neither DICT nor DATA is specified, CLEARFILE will clear the DATA portion of the file.

filename

Filename is the name of the file to be cleared. You may clear any attached file.

option

Use the (S) option (enclosed in parentheses) to suppress the display of messages.

Correct Use of CLEARFILE Command

```
CLEARFILE DICT ORDERS
```

All the records in the dictionary of the ORDERS file will be deleted.

```
CLEARFILE DATA ORDERS
```

All the records in the data portion of the ORDERS file will be deleted.

```
CLEARFILE STAT
```

All the records in the data portion of the STAT file will be deleted.

CLEARLIST

The R/LIST command CLEARLIST discards an active list of record keys.

CLEARLIST is a synonym for CLEAR.

CLEARLIST**Using CLEARLIST**

Use CLEARLIST to deactivate a filter (a list of record keys) in the current environment.

A filter can be made active by a previously executed GETLIST, MAKELIST, SELECT, or SSELECT command.

- ✎ CLEARLIST affects only the filter active in RAM. It has no effect on lists stored on file.

Correct Use of CLEARLIST Command

```
GETLIST JAN
```

```
CLEARLIST
```

The list JAN is no longer available to the next process.

COMMITLOG

The COMMITLOG command displays all transactions that are currently being committed.

```
COMMITLOG
```

Using COMMITLOG

The COMMITLOG report displays the:

- User name (and station ID for networked versions)
 - File name
 - Commit queue (unless the Environment window's *Commit Protection* prompt is set to OFF)
- ☛ If necessary, the SYSPROG user (system administrator) may remove transaction entries from the commit queue. See the “Managing Files and Records” chapter of *Filing Systems*.

Correct Use of COMMITLOG Command

```
COMMITLOG
```

A report of the current user's committed transactions is displayed.

COMPILE

The COMPILE command turns R/BASIC source code into executable object code.

```
COMPILE filename keys {options}
```

Using COMPILE

COMPILE translates R/BASIC source code into R/BASIC object code. If the compilation is successful, the compiler writes the R/BASIC object code into the same file that contains the source code.

- ☛ R/BASIC programs are records in an Advanced Revelation file. The programs can be up to 64K in length.

As the compiler works, it prints an * (asterisk) for each line of source code compiled. If the compilation is not successful, Advanced Revelation displays appropriate error messages, and the object code record is not written to a file. If the compilation is successful, Advanced Revelation will display the message:

```
Line (n) [B0] Compilation completed
```

filename

Filename is the name of the file containing the record(s) to compile.

keys

Keys is a list made up of one or more record keys (program names) separated by spaces. Substitute an * (asterisk) as the record key to compile all the records in a file. To perform COMPILE using an active list of record keys, the *keys* list must be empty.

options

Options must be enclosed within parentheses. More than one option can be used within a single set of parentheses. Six options are available.

Option	Meaning
(B)	Use B to compile only R/BASIC code. Any embedded SQL code is not compiled.
(C)	Use C to cut off the symbol table (table of variables) for the object code, thus saving space. If the C option is used, the debugger cannot report the value of variables.
(L)	Use L to suppress linemarks in the object code. The object code is shortened one character for each linemark suppressed, thus taking less disk space and executing slightly faster. However, the Debugger can only break into a program at a linemark. If LINEMARK statements are inserted into the source code prior to compilation, the Debugger line count will not be accurate.
(P)	Use P to print compiler messages to the printer.
(S)	Use S to suppress the display of asterisks and the system message that indicates successful compilation. Compilation error messages still display, and if the S option is used with the P option, messages will be sent to the printer.
(X)	Use X to capture any SQL source code to file (see the "Programming with SQL" chapter in <i>SQL</i>).

Correct Use of COMPILE Command

```
COMPILE INV.BP PGM456 PGM789
```

This command will compile the source code for the records PGM456 and PGM789 in the INV.BP file and write the resultant object code to the INV.BP file under the names \$PGM456 and \$PGM789.

```
COMPILE INV.BP * (P)
```

This command will COMPILE all records in the INV.BP file. Error messages are routed to the printer.

```
SELECT BP WITH @ID "INV]"
COMPILE BP
```

This command will COMPILE the selected records. Notice that the selection criterion specifies that only those records in the BP file containing the character string INV in the field @ID be selected.

CONTROL_OFF

The CONTROL_OFF command removes the transaction and domain controls from a file. The controls must have previously been added to the file with CONTROL_ON.

- For information about transaction and domain controls, see the “Files and Records” chapter of *Filing Systems*.

```
CONTROL_OFF {volume} {filename ::} {options}
```

volume

The name of an Advanced Revelation volume. If *volume* and *filename* are not specified, a selection popup is displayed of all currently attached files having control features.

filename

The name(s) of the data file(s) in *volume* from which to remove controls. If *volume* is specified and *filename* is not, a selection popup is displayed of all files in *volume* that have control features. If *filename* does not exist in *volume*, an error message is displayed.

options

The options are enclosed in parentheses. More than one option can be used within a single set of parentheses. Three options are available:

Option	Meaning
(S)	Suppress display. A selection popup is not displayed. The CONTROL_OFF operation is not executed if neither <i>volume</i> nor <i>filename</i> is specified.

Option	Meaning
(T)	Temporary. Control features are removed, and automatically applied again when the file(s) is reattached or attached by a new user. Any files specified in <i>filename</i> must be currently attached.
(R)	Report. Displays the number of files affected by the CONTROL_OFF operation (even if the S option is used).

Correct Use of CONTROL_OFF Command

CONTROL_OFF INVENTORY NEW_PRODUCTS
Removes control features from the NEW_PRODUCTS file in the INVENTORY volume.

CONTROL_OFF INVENTORY
Displays a selection popup of all files in the INVENTORY volume that have control features.

Incorrect Use of CONTROL_OFF Command

CONTROL_OFF INVENTORY CUSTOMERS
An error message is displayed if the CUSTOMERS file does not exist in the INVENTORY volume.

CONTROL_ON

The CONTROL_ON command adds transaction and domain controls to a file.

☞ For information about transaction and domain controls, see the “Managing Files and Records” chapter of *Filing Systems*.

```
CONTROL_ON {volume} {filename ::} {options}
```

volume

The name of an Advanced Revelation volume. If *volume* and *filename* are not specified, a multiselection popup is displayed of all currently attached files that do not have control features.

☞ By pressing [Enter] at any of the highlighted selections displayed in the popup, you can add or remove transaction controls.

filename

The name(s) of the data file(s) in *volume* to which controls are to be added. If *volume* is specified and *filename* is not, a selection popup is displayed of all files in *volume* that do not have control features. If *filename* does not exist in *volume*, an error message is displayed.

- ☞ By pressing [Enter] at any of the highlighted selections displayed in the popup, you can add or remove transaction controls.

options

The options are enclosed in parentheses. More than one option can be used within a single set of parentheses. Three options are available:

Option	Meaning
(S)	Suppress display. A selection popup is not displayed. The CONTROL_ON operation is not executed if neither <i>volume</i> nor <i>filename</i> is specified.
(T)	Temporary. Control features are automatically removed when the file(s) is reattached or attached by a new user. Any files specified in <i>filename</i> must be currently attached.
(R)	Report. Displays the number of files affected by the CONTROL_ON operation (even if the S option is used).

Correct Use of CONTROL_ON Command

```
CONTROL_ON INVENTORY NEW_PRODUCTS
```

Adds control features for the NEW_PRODUCTS file of the INVENTORY volume.

```
CONTROL_ON INVENTORY
```

Displays a selection popup of all files in the INVENTORY volume that do not have control features.

Incorrect Use of CONTROL_ON Command

```
CONTROL_ON INVENTORY CUSTOMERS
```

An error message is displayed if the CUSTOMERS file does not exist in the volume INVENTORY.

COUNT

The R/LIST command COUNT totals the number of records in a file that meet the condition(s) specified.

```
COUNT {DICT} filename {USING alt_filename}{selection_criteria}⇒
      {options}
```

DICT

Use DICT to count only the number of fields in the dictionary of the file.

filename

Use *filename* to specify any attached Advanced Revelation file. If you specify only the filename, the program totals the number of records in the file.

If there is a list of active record keys, the program counts the total number of record keys in the list.

USING

A USING clause specifies that the COUNT operation is to use the dictionary of an alternative file, rather than the *filename* dictionary.

<i>alt_filename</i>	The name of the alternative file (not the dictionary file name).
---------------------	--

☞ See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria defines the specific conditions to be met before the program counts any records. For example, suppose you want to know how many customers live in the state of Washington:

```
COUNT CUSTOMERS WITH ST EQ 'WA'
```

For more information about *selection_criteria*, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

options

Options must be enclosed within parentheses. More than one option can be used within a single set of parentheses. Two options are available.

Option	Meaning
(P)	Use the P option to send the total number of records counted to the printer.
(S)	Use the S option to suppress message output.

Correct Use of COUNT Command

COUNT CUSTOMER WITH STATUS "A"

Counts the records in the CUSTOMER file having A in the STATUS field.

COUNT INVENTORY

Counts the records in the INVENTORY file.

COUNT INVOICES WITH INV.TOT > 500

Counts the records with over \$500 on an invoice.

CW

CW displays the specified command window.

CW window

window

The name of a command window having a template in the COMMANDS file.

Correct Use of CW Command

CW BATCH

Displays the Batch window.

CW RECORDCOPY

Displays the RECORDCOPY window.

DEFINEFILE

The DEFINEFILE command creates and attaches a file.

MAKEFILE is a synonym for DEFINEFILE.

```
DEFINEFILE {vol} [{DICT}] filename {attributes} ⇨
               [{DATA}]
               {FIELDS = [field_def ::]} {option}
```

Using DEFINEFILE

Use DEFINEFILE to add an entirely new file into the current account.

- ☞ If you use DEFINEFILE on a network, other network users will need to ATTACH the volume used for the new file before they will be able to access it.

vol

Vol (volume) defines the complete directory path, including the drive specifications. Use a colon as part of the drive specification. *Vol* can also specify a volume record in the VOC file (See SETVOLUME).

If *vol* is not specified, the program creates a new file on the default volume. If *vol* is not attached, the program attaches the volume and updates it when it creates the new file. If the REVVOL map does not exist, the program creates one.

DICT

Use DICT to specify that only the dictionary file be made.

DATA

Use DATA to specify that only the data file be made.

- ☞ If DICT and DATA are not specified, the program makes both files.

filename

A valid file identifier.

- ☞ For more information, see the “Files and Records” chapter of *Filing Systems*.

attributes

The *attributes* you may specify vary according to the filing system or bonding. Refer to the “Files and Records” chapter of *Filing Systems*, or to the bond documentation for the appropriate specifications.

- ☞ The Definefile window prompts for the appropriate *attributes* specifications for your filing system or bond.

FIELDS

The FIELDS keyword assigns field definitions for *filename*. A field definition consists of the:

- Name of the field
- Data type specification for the field
- Optional keyword KEY, defining this field as a primary key field for *filename*. At least one field in *filename* must be a primary key field.

Multiple field definitions are delimited by commas, and the list is enclosed in parentheses.

field_def The field definitions for *filename*

option

Use the (S) option (enclosed within parentheses) to suppress message output.

Correct Use of DEFINEFILE Command

```
DEFINEFILE C:\REV CUST
```

A file named CUST is created on the volume named \REV located on Drive C. The statement will create both data and dictionary files.

```
DEFINEFILE CUSTOMERS FIELDS = (CUST_NO INTEGER KEY, FNAME
CHAR(12), LNAME CHAR(35), ADDRESS VARCHAR, CITY CHAR(25), ST
CHAR(3))
```

The CUSTOMERS file is created on the default volume with six fields defined. CUST_NO is defined as the primary key field.

DELETEACCOUNT

The DELETEACCOUNT command permanently deletes an account and all the files associated with it.

DELETEACCOUNT is a synonym for DELETE-ACCOUNT.

DELETEACCOUNT account

Using DELETEACCOUNT

DELETEACCOUNT is used to remove an account previously established with the MAKEACCOUNT command. DELETEACCOUNT removes all files associated specifically with the account (but no global files) and removes the appropriate entries in the SYSTEM file for the account pointer, as well as environment and color settings.

DELETEACCOUNT will initially remove all files on the default volume for the account being deleted. If files for this account also reside on other volumes, the user can specify these volume names as the DELETEACCOUNT process proceeds.

DELETEACCOUNT can only be executed from the SYSPROG account.

Any Qfile pointers that are pointing to files in the deleted account can be manually detached, or will be detached automatically when the user logs off.

- ☛ If you use DELETEACCOUNT on a network, be sure no user is logged onto that account.

account

Account specifies which account to delete. DELETEACCOUNT removes all files associated specifically with the account (but no global files).

- ☛ There may be a Qfile pointer to a file in the deleted account. If so, detach the file. To do this, either use the DETACH command or log off and then log back on.

Correct Use of DELETEACCOUNT Command

```
DELETEACCOUNT TEST
```

The TEST account and the associated files and records are removed from the system. Qfile pointers that point to files in the deleted account are not removed.

DELETEFILE

The DELETEFILE command permanently removes a file from the system.

- ☛ For information about deleting a bonded file, see the documentation for the bond.

```
DELETEFILE {DICT} {DATA} filename {option}
```

Using DELETEFILE

Use DELETEFILE to remove files that are no longer needed on the system. The space previously occupied by the file will be made available on disk. All indexes associated with the data file are also deleted.

- ☛ If the No-Delete flag in the Definefile window has been set for the file, the data portion of the file will not be deleted.

You will be prompted to confirm that you wish to delete the file.

If you use DELETEFILE on a network, other network users will need to ATTACH the volume from which the file was deleted in order to update their FILES file.

DICT

Use **DICT** before the file name to delete only the dictionary portion of the file. If not specified, both the dictionary and data portions are deleted.

DATA

Use **DATA** before the file name to delete only the data portion of the file. If not specified, both the dictionary and data portions are deleted.

filename

Filename is the name of the file to be deleted. It must be attached. **DELETEFILE** removes all Advanced Revelation and operating system references to the file. Do not specify a Qfile as a filename.

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of DELETEFILE Command

```
DELETEFILE TEMP
```

The TEMP file is deleted from the system.

DELETEREC

The **DELETEREC** command permanently removes record(s) from an Advanced Revelation file.

DELETEREC is a synonym for **DELETE**.

```
DELETEREC {DICT} filename key {option}
```

Using DELETEREC

Use **DELETEREC** to remove individual records from a file. Use with care! The system does not prompt you to confirm the command. Once a record is deleted from a file, it cannot be recalled.

- ✎ A Delete Record command in some non-Revelation systems preserves a deleted record's file space by inserting a mark that the program reads as "record deleted." **DELETEREC** does not do this. The program completely removes the record and releases the space as soon as the command is executed.

DICT

Use **DICT** to delete records from the dictionary portion of a file.

filename

Use only an attached Advanced Revelation file.

keys

Keys is a list made up of one or more record keys, separated by spaces. The *keys* parameter is required, except when deleting records while using an active list of record keys. Substitute an * (asterisk) for *keys* to delete all the records in the file.

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of DELETEREC Command

```
DELETEREC CUSTOMER 101
```

This command deletes record 101 from the CUSTOMER file.

```
DELETEREC DICT CUSTOMER *
```

This command deletes all dictionary items in the CUSTOMER file.

```
GETLIST SAMPLE  
DELETEREC CUSTOMER
```

The records from the CUSTOMER file that exist in the list of record keys named SAMPLE are deleted.

DELETEUSER

DELETEUSER permanently removes a user name from the system.

```
DELETEUSER user {option}
```

Using DELETEUSER

Use DELETEUSER to remove a user name from the system. All references in the SYSTEM file to the user (including environment and video settings) are permanently removed.

✎ If using DELETEUSER on a network, be sure no one is logged in under the *user* name.

For more information about users, accounts, and security, see the chapter “Users and Accounts: Planning Access and Security” in *Developing an Application*.

user

User specifies the name to remove from the system.

option

Use the (S) option (enclosed in parentheses) to suppress system messages.

Correct Use of DELETEUSER Command

```
DELETEUSER MIKE
```

This command permanently deletes the user called MIKE, and removes the associated environment and video settings.

DELLIST

The R/LIST command DELLIST permanently removes a list of record keys.

```
DELLIST {listname} {option}
```

Using DELLIST

Use DELLIST to remove lists of record keys that are no longer current. Deleting old lists of record keys frees up space on file.

listname

Listname is a list of record keys previously saved with SAVELIST. If *listname* is not specified, the program deletes the record '' (null).

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of DELLIST Command

```
SELECT CUSTOMERS WITH ZIP OVER 10000
```

The SELECT command creates a list of record keys that are saved with the SAVELIST command.

```
SAVELIST SAMPLE  
GETLIST SAMPLE  
LIST CUSTOMERS  
DELLIST SAMPLE
```

GETLIST is used to retrieve and activate the list of keys. The list is then used as input by the LIST command. Once LIST has finished, the DELLIST command is used to delete the saved list.

DETACH

The DETACH command removes the availability of Advanced Revelation files or volumes to the system.

```
DETACH {volume} {DICT} {DATA} file
```

Using DETACH

Files that have been previously attached are available for processing in the system. The list of currently attached files is stored in the system file FILES. Use DETACH to remove one or more files from the FILES file or to detach an entire volume. The files will then be inaccessible until reattached.

☞ DETACH will not operate on the following files:

COMMANDS	VERBS	DICTVOC	VOC
MESSAGES	FILES	VOLUMES	

volume

The name of the volume to detach. You may optionally specify a volume record in the VOC file (see the SETVOLUME command).

If neither *volume* nor *file* is specified, all files on the default volume are detached (except for those system files listed above).

DICT

Specify DICT to detach only the dictionary for *file*. The default operation detaches both the DICT and DATA portions of *file*.

DATA

Specify DATA to detach only the data file. The default operation detaches both the DICT and DATA portions of *file*.

file

The currently attached file or files to detach. If *file* is not specified, all files on *volume* are detached.

Correct Use of DETACH Command

```
DETACH B:
```

Files attached on volume B: will no longer be accessible.

```
DETACH
```

All files on the default volume (except the restricted system files) are detached.

DICT

The DICT command displays the DICT command window, which creates, modifies, and deletes dictionary records. A dictionary record stores one of three kinds of information: field specifications, an R/BASIC formula, or an R/LIST modifying phrase.

```
DICT {filename} {fieldname}
```

Using DICT

Use DICT to build dictionary records. There are three types of dictionary records:

- Type F is a field definition record. A type F dictionary record refers to a specific field in a data record and defines how the data in that field is used by windows and reports.
- Type S is a symbolic dictionary record. This type of record contains a formula of R/BASIC rules that define the product yielded by the dictionary record when it is used.
- Type G is a group dictionary record. This type of dictionary record uses the contents of Type F or Type S fields to construct a phrase that modifies an R/LIST command.

The prompts for this window differ for each type of dictionary record.

filename

The *filename* specifies the file with which the dictionary records are associated.

fieldname

The *fieldname* specifies the name of the dictionary field.

Correct Use of DICT Command

```
DICT CUSTOMER COMPANY
```

The command displays the DICT window and loads CUSTOMER and COMPANY in the file name and field name prompts.

DUMP

The DUMP command allows you to examine the contents of an Advanced Revelation Linear Hash file frame by frame.

```
DUMP {DICT} filename
```

Using DUMP

Use DUMP to examine the internal structure of a Linear Hash file. DUMP will display the file frame by frame. This enables you to see the physical layout of data in the file and to fix group format errors.

- ☞ This program is for use by advanced users only. Novice users should use DUMP only under the direction of Advanced Revelation Technical Support.

Warning ! DUMP can make sweeping changes to the contents and structure of Linear Hash files, and should only be used with extreme caution. Always back up files before modifying them with DUMP.

Options available through DUMP include:

- Changing the threshold percentage at which a group will be split
- Setting or clearing the sizelock
- Changing individual bytes in the data of the frame
- Finding and fixing group format errors

DICT

Use DICT to DUMP only the dictionary portion of the file.

filename

The file specified by *filename* must be an Advanced Revelation Linear Hash file.

Cursor Control

Use these keys to move around within the file.

- [→] The right cursor arrow moves the display to the next frame or to the overflow frames of a group.
- [←] The left cursor arrow moves the display to the previous frame.
- [↓] The down cursor arrow moves the display to the next group.
- [↑] The up cursor arrow moves the display to the previous group.

Correct Use of DUMP Command

DUMP CUSTOMERS

Displays frame one of the CUSTOMERS file.

EDIT

The EDIT command invokes the Editor, which allows Advanced Revelation records and operating system files to be created, viewed, modified, or deleted.

```
EDIT {DICT} {DATA} filename keys
```

Using EDIT

EDIT invokes the Advanced Revelation Editor.

DICT

Use DICT to edit record(s) in the dictionary of *filename*.

filename

The name of any attached file.

keys

Key is a list made up of one or more record keys separated by spaces. Substitute an * (asterisk) as the key to edit all the records in a file. To perform EDIT using an active list of record keys, the key list must be empty.

✎ For details on how to use the Editor, see *Getting Started*.

Correct Use of EDIT Command

```
EDIT ORDERS 123
```

Edit record 123 in the ORDERS file.

```
EDIT BP *
```

Edit all records in the BP file.

ENCRYPT

The ENCRYPT command uses an irreversible encryption algorithm to restrict access to a template record.

```
ENCRYPT template_file records {TO: {new_file} ⇨  
  {new_records}} {options}
```

template_file

The name of the file containing the templates to encrypt and copy to *new_file*.

records

The template records to encrypt and copy. Use * (asterisk) to encrypt all records. The source records will not be affected by the encryption process unless the records are copied to the source file.

Warning ! The encryption process is not reversible. Encrypted records cannot be unencrypted.

new_file

The name of the file to which the encrypted records will be copied. The file name is preceded by a parenthesis. If *new_file* is not specified, the (N) or (O) option must be used to write the records back to the *template_file*.

new_records

The names to be assigned to the encrypted records. If *new_records* is not specified, the source record names are retained.

options

The options are enclosed in parentheses. The following options are available:

Option	Meaning
(N)	Copy and overwrite only if record exists
(O)	Overwrite only if record exists
(S)	Suppress the display of messages

Correct Use of ENCRYPT Command

ENCRYPT TEMPLATES LIBRARY TO: (ENCRYPTED_TEMPLATES
The *LIBRARY* template is encrypted and copied to the *ENCRYPTED_TEMPLATES* file.

ENCRYPT TEMPLATES * (O)
All templates are encrypted. The encrypted templates are copied back to the *TEMPLATES* file, overwriting the existing template records.

EVAL

The EVAL command compiles and executes the specified R/BASIC source code.

To execute an R/BASIC program from TCL, use the RUN command.

EVAL *code*

code

The *code* may be any R/BASIC source code. Multiple lines are delimited by & (ampersand).

Correct Use of EVAL Command

```
EVAL CALL MSG("A Sample Message", "", "", "")
```

A call to the system subroutine MSG is compiled and executed.

Incorrect Use of EVAL Command

```
EVAL TEST_PROG
```

A program name cannot be used.

EW

The R/LIST command EW invokes the EasyWriter report-writing utility.

EW

Using EW

Entering EW calls EasyWriter.

- ✎ For information about the EasyWriter utility, see the chapter “Writing Reports with EasyWriter” in *Creating Reports*.

Correct Use of EW Command

```
EW
```

Displays the EasyWriter window and menu.

EXPORT

EXPORT uses conversion process defined previously in the Export window to convert an Advanced Revelation file to a foreign file format.

- ☞ This is not an environmental bond. For information about Environmental Bonding, see the chapter “Managing Files and Records” in *Filing Systems*.

EXPORT format process

Using EXPORT

Use EXPORT to convert an Advanced Revelation file to the desired format and write the data to an operating system file.

format

An available conversion format. The formats currently supported are:

- ASCII
- LOTUS 123
- dBASE III

process

The name of a conversion process defined at the appropriate Export window (Export ASCII or Export LOTUS 123, for example). A defined conversion process is saved as a record in the IMPORT.EXPORT file.

- ☞ *Process* is not the extended process name assigned to the record in the IMPORT.EXPORT file, but the conversion process name that you specify at the Export window.

Correct Use of EXPORT Command

EXPORT ASCII CHECK

The record CHECK is the name of a conversion process previously defined at the appropriate Export window. The data is converted to ASCII format and written to an operating system file.

FILECOPY

FILECOPY transfers an attached file to any account in any volume. Use it to relocate and/or duplicate a file.

```
FILECOPY {DICT} {DATA} filename TO: {vol} {account} ⇨  
        {new_filename} {options}
```

Using FILECOPY

Use FILECOPY to transfer files:

- From one volume to another
- From one volume to another while renaming the file
- To the same volume under a different name
- From the default account to another account
- From the default account to another account while renaming the file

Any information on the right side of **TO:** is processed in the order of the above parameters. In other words, the first word to the right side of **TO:** is assumed first to be a volume designation. If no such volume exists, it is assumed to be an account designation. If the account does not exist, it is assumed to be a file name. To ensure that there is no confusion between these parameters, it is safest to specify all three.

- ✎ Use FILECOPY to copy a file to another volume using the same filing system. Use REDEFINEFILE to copy a file to a volume with a new filing system.

DICT

Use DICT to copy only the file's dictionary.

DATA

Use DATA to copy only the file's data file.

- ✎ If you do not specify DICT or DATA, FILECOPY copies both.

filename

Use *filename* to define the file you want to transfer. You may use any attached Advanced Revelation file. The program copies only one file at a time. If the filename does not exist, the program displays a message.

vol

Vol (volume) specifies the target volume for the copy process.

account

Use *account* to specify a destination account for the new file. If not specified, the program copies the file to the same account.

new_filename

New_filename specifies the new name of the copied file.

options

Options must be enclosed within parentheses. Four options are available.

Option	Meaning
(D)	Use D to delete the source file after successfully copying a record to the destination file.
(N)	Use N to write the source file over the destination file, only if one already exists. The N option generates an error message if the destination file does not exist.
(O)	Use O to write the source file over the destination file. If the destination file does not exist, a new file is created.
(S)	Use S to suppress message output.

Correct Use of FILECOPY Command

FILECOPY CUSTOMERS TO: CUST.DATA CUSTOMERS (D)
This command copied the file CUSTOMERS to the CUST.DATA volume under the same file name. Once the copy is complete, the original file is deleted.

FILTER

The R/LIST command FILTER executes a previously defined filter (SELECT statement).

FILTER filter_name

Using FILTER

FILTER runs a filter template defined at the Filter window. The resulting list of keys (if any) is then available to the next process.

The filter template is stored in the FILTERS file.

filter_name

The name of a filter stored as a record in the FILTERS file.

Correct Use of FILTER Command

FILTER MONTHLY

The filter MONTHLY executes to return a list of record keys.

FLUSH

The FLUSH command writes to disk any Advanced Revelation data currently stored in memory.

FLUSH

Correct Use of FLUSH Command

FLUSH

Any Advanced Revelation data in memory is written to disk.

FORM

The FORM command creates a report based on previously defined report form specifications.

FORM {filename} template key_list {option}

Using FORM

FORM will execute and display a report based on specifications painted previously using the Form window. FORM is used to merge data from a file into the template specified in the command.

For more information about creating templates to be used with FORM, see the chapter “Creating Forms” in *Creating Reports*.

filename

The name of a file in which the specified template is stored (defaults to REPORTS). The file name must be prefixed with @ (at-sign).

template

The name of a form template stored in the specified file. This template must have been previously created using the form creation window.

key_list

Key_list is a list made up of one or more record keys separated by spaces. Substitute an * (asterisk) as the key to list all the records in a file. To perform FORM using an active list of record keys, the key list must be empty.

option

Use the (P) option (enclosed within parentheses) to route the report to a printer.

Correct Use of FORM Command

```
FORM INVOICE.RPT * (P)
```

This command will create a report based on the template INVOICE.RPT stored in the REPORTS file. All records from the data file will be used in the report, and the report will be printed on the printer.

GETLIST

The R/LIST command GETLIST retrieves a saved list of record keys for processing.

GETLIST is a synonym for GET-LIST.

GETLIST listname

Using GETLIST

Use GETLIST to retrieve a list of record keys created using SELECT or SSELECT and stored using SAVELIST. GETLIST makes this list of record keys active for the next command.

☛ For more information about lists of active record keys, see “Using Filters” in the “Introduction to R/LIST” chapter of *R / LIST*.

listname

Listname is the name of a list of record keys previously stored with a SAVELIST command.

Correct Use of GETLIST Command

```
SSELECT CUSTOMERS WITH ST 'WA'
SAVELIST SAMPLE
GETLIST SAMPLE
LIST CUSTOMERS
```

The SSELECT command creates a list of record keys that is written to the LISTS file using SAVELIST. The list is retrieved by GETLIST and then is ready for processing by LIST.

IMPORT

IMPORT uses a conversion process previously defined in the Import window to convert a foreign file to Advanced Revelation file format.

- ✎ This is not an environmental bond. For information about Environmental Bonding, see the chapter “Managing Files and Records” in *Filing Systems*.

IMPORT format process

Using IMPORT

Use IMPORT to convert a foreign file from an existing format to Advanced Revelation format. The data is written to an Advanced Revelation file specified at the Import window.

format

An available conversion format. The formats currently supported are:

- ASCII
- LOTUS 123
- dBASE III

process

The name of a conversion process defined at the appropriate Import window (Import ASCII or Import LOTUS 123, for example). A defined conversion process is saved as a record in the IMPORT.EXPORT file.

- ✎ *Process* is not the extended process name assigned to the record in the IMPORT.EXPORT file but the conversion process name that you specify at the Import window.

Correct Use of IMPORT Command

```
IMPORT ASCII CHECK.DOC
```

The operating system file CHECK.DOC is converted from ASCII format and written to an Advanced Revelation file.

INDEX

The INDEX command updates an existing index for one field in a file.

INDEX filename field index_type

Using INDEX

Use INDEX to update the index for one field in a file. It may be necessary to use INDEX if an index control record has been created but the index was never actually updated.

INDEX assumes that an index control record has already been created. The index control record determines which type of indexing has been established for the field. Index control records are created using the windows available from the Index menu.

For more information about indexing, see *Accessing Data*.

filename

The *filename* is the name of the file that is being indexed. Use any attached file that is flagged as being indexed. The keyword ALL can be used here to indicate that all files are to be updated.

field

The *field* is the name of the field in *filename* that is being indexed. The field must previously have been flagged as being an indexed field. The keyword ALL can be used here to indicate that all fields in the specified file (or in all files if ALL is specified for *filename*) are to be updated.

index_type

The *index_type* is one of three types of indexing supported: Btree, Crossref, and Relational. Fields can be indexed using more than one type of *index_type*, but INDEX will only update one type of index at a time. INDEX assumes that the *index_type* has already been established using the index control window(s).

Correct Use of INDEX Command

```
INDEX CUSTOMERS COMPANY BTREE
```

This command updates the index for the field COMPANY in the CUSTOMERS file. The index is a Btree index.

LABEL

The LABEL command executes a report based on previously-defined mailing label parameters.

```
LABEL label_name {options}
```

Using LABEL

LABEL executes a report based on parameters previously established with the MAKELABEL window. Use LABEL when you wish to run a sequence of mailing labels that you have already set up.

LABEL reads a label definition record from the LABELS file. To change parameters for the labels, use the MAKELABEL window to edit the label definition record.

LABEL will process all records in the file contained in the label definition record. If a filter is active, only those records represented by the active list will be processed.

label_name

The *label_name* is the name under which the label definition record was stored.

options

Options must be enclosed within parentheses. Two options are available:

- (P) Use P to print labels to the printer.
- (T) Use T to print labels to the screen (terminal).

Correct Use of LABEL Command

```
SELECT CUSTOMERS WITH ST = "WA"
LABEL CUSTOMER.LABEL (T)
```

In this example, a SELECT command is used to create an active list of all customers in the state of Washington. The LABEL command then creates a report of those customers based on the format stored in the label definition record CUSTOMER.LABEL. The report is routed to the screen by the T option.

LIST

The R/LIST command LIST creates a columnar report out of data in an Advanced Revelation file. LIST includes options for selecting, sorting, and displaying records in the file.

```
LIST {n} {ONLY} {DICT} filename {USING alt_filename} ⇨
      {selection_criteria} {sort_criteria} {fields} ⇨
      {keywords} {options}
```

Using LIST

Use LIST to produce formatted output from an Advanced Revelation file. LIST creates reports in which each record is printed as a row; fields from the record are printed in columns.

For more information about the use of LIST, see the *R/LIST* section.

n

Use *n* to specify the number of records to list. The resulting list will be a random selection of the first *n* records in the file.

ONLY

Use ONLY to use only record keys in the report or form. LO is a synonym for LIST ONLY. It takes no selection or sort criteria.

DICT

Use DICT to list only the dictionary portion of the file.

filename

Use *filename* to specify any attached Advanced Revelation file.

USING

A USING clause specifies that the report or form is to use the dictionary of the operating system file rather than the *filename* dictionary.

alt_filename The name of the alternate file (not the dictionary file)

See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria enables you to indicate a subset of records to display by including filtering specifications in the LIST command.

For more information about *selection_criteria*, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

sort_criteria

Sort_criteria enables you to specify the order by which records are to be displayed.

For details about *sort_criteria*, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

fields

Fields designates the fields that are to be displayed for each record. The fields should be included in the LIST command in the order in which the fields are to be displayed across the page, left to right.

If no fields are specified, LIST will use the list of field names kept in the dictionary records @CRT (for screen output) or @LPTR (for printer or disk output). If neither of these dictionary records exists in the dictionary of *filename*, LIST will use the field names defined in the @CRT or @LPTR records in the dictionary of the VOC file.

keywords

The *R/LIST keywords* include control, formatting, and output specifications. For details on the use of these keywords, see the “R/LIST Keyword Reference” chapter of the *R/LIST* section.

options

The following options are available and must be enclosed in parentheses.

Option	Meaning
(E)	Use E to suppress the initial top-of-form when printing to the printer.
(N)	Normally, when the output goes to the screen, the program waits at the end of each page. Use N if you want the report to scroll continuously.
(P)	Use P to output the printer.
(X)	Use X to cause LIST to save generated R/BASIC code to the operating system file RLIST.
(SX)	Use SX to cause LIST to save generated R/BASIC code to the operating system file RLIST but not execute the report.

Correct Use of LIST Command

LIST CUSTOMERS

This will produce a report of customers and display the default fields specified by the @CRT record in the dictionary.

LIST CUSTOMERS (P)

This will produce the customer report and display the default fields specified by the @LPTR record in the dictionary.

LIST ORDERS IF ANY INV.AMT GE "100" ORDER.NO INV.DT TOTAL INV.AMT

This will produce a report of orders with invoice amounts greater than or equal to \$100. The report will total the invoice amount field.

LIST DICT ORDERS BY F1 HEADING "ORDERS FILE LAYOUT PAGE 'PLL'"

This will produce a report of the dictionary of the ORDERS file sorted by the first field. The report's heading will show ORDERS FILE LAYOUT PAGE and will insert a current page number followed by two blank lines.

LISTDICT

LISTDICT displays the contents of a file’s dictionary.

`LISTDICT filename {options}`

Using LISTDICT

LISTDICT produces a report of the dictionary records of a file. Each of the fields below is listed on the report:

- Field Name

The name used to access the field
- T

Indicates the type of field. This can be “F” for true data fields, “S” for symbolic fields (defined by an R/BASIC formula), or “G” for groupings of other fields.
- Pos

A number identifying the position of the field in the record. The field number is used to extract the data for that field from the record. Dictionary records that contain a formula do not have a field number. The key to the file is position 0.
- S/M

Indicates a single or multivalued field.
- Data Type

The type of data required for a field.
- Just

The type of justification for data in the field. There are four options:

Character	Meaning
R	Right justification (used with numeric fields).
L	Left justification (used with alphabetic fields).
C	Center justification (to enter data within the specified length).
T	Text (paragraph) justification (used for long text fields).

Len The number of characters that will be output when data from this field is used on a report.

⚠ Neither Just nor Len has any effect in the format of data stored on file. They are used strictly when creating reports.

filename

The *filename* specifies any attached Advanced Revelation file name.

options

Options must be enclosed within parentheses. Two options are available:

Option	Meaning
(F)	Sort by field name.
(N)	Do not sort output.
(Pn)	Send output to the printer designated by the number <i>n</i> .

Correct Use of LISTDICT Command

```
LISTDICT CUSTOMER
```

Displays a report of the CUSTOMER dictionary.

LISTFILES

The LISTFILES command lists the files that are currently attached.

```
LISTFILES {options}
```

Using LISTFILES

Use LISTFILES to list all files in the FILES file, regardless of what volume they were attached from. LISTFILES displays the following information:

File Name	The name displayed is the Advanced Revelation name of the file.
Account	The account with which the file is associated.
Volume Name	The name of the volume where the file is located.
File Type	The type of file as determined by the filing system.

options

The following options are available (enclosed within parentheses):

Option	Meaning
(A)	Display system files as part of the list.
(F)	Sort the list by file name.
(N)	Do not sort the list.
(O)	Display records specified by another option.
(P <i>n</i>)	Send the report to the printer designated by the number <i>n</i> .
(U)	Display files used this session at the top of the list.
(X)	Display Qfiles at the top of the list.

Correct Use of LISTFILES Command

```
LISTFILES (A)
```

Displays all the files attached including the system files.

LISTINDEX

The LISTINDEX command displays file and field indexing information for Advanced Revelation files.

```
LISTINDEX {filename} {fieldname} {option}
```

Using LISTINDEX

Use LISTINDEX to display indexing information for a field, a file, or all files with indexing.

When LISTINDEX is used without *filename* and *fieldname*, a popup displays a list of all Advanced Revelation files that are indexed.

filename

The *filename* is the name of an attached file containing the indexing information to list. A popup displays the indexing information for the specified file.

fieldname

The *fieldname* is the name of a dictionary record with indexing information. Using *fieldname* calls a window that displays the indexing information for that field name.

option

Use the (P) option (enclosed in parentheses) to route the report to the printer.

Correct use of LISTINDEX Command

```
LISTINDEX CUSTOMER (P)
```

The index report for the CUSTOMER file is routed to the printer.

LISTVOL

The LISTVOL command lists all attached and unattached files in a volume.

LISTMEDIA is a synonym for the LISTVOL command.

```
LISTVOL {volume} {option}
```

Using LISTVOL

Use LISTVOL to determine which files exist on any given volume. LISTVOL will list the contents of the volume's directory, whether or not any files from that volume have been attached.

LISTVOL displays the following information:

File Name	The Advanced Revelation name of the file.
Account	The account name with which the file is associated.
OS/Foreign Name	The operating system file name.
MFS	A modifying filing system used for special purposes, such as indexing or transaction control.

volume

Use *volume* to indicate the volumes for which you wish to list the volume directory. If no volume is indicated, LISTVOL will list the files on the default volume.

- For more information about volumes, see the chapter "Advanced Revelation Volumes" in *Filing Systems*.

options

The following options are available and must be enclosed within parentheses:

Option	Meaning
(F)	Sort the list by file name.
(N)	Do not sort the listing.
(P <i>n</i>)	Send the report to the printer designated by the number <i>n</i> .

Correct Use of LISTVOL Command

LISTVOL
Displays a list of files on the default volume DATAVOL.

LISTVOLUMES

LISTVOLUMES creates a report of all volumes that are currently attached.

LISTVOLUMES {options}

Using LISTVOLUMES

Use LISTVOLUMES to create a report of all volumes that are currently attached. The report is based on the contents of the VOLUMES file.

The LISTVOLUMES report displays the following information:

Volume Name	The name by which the volume was attached (a drive or path designation or a volume name as created by SETVOLUME).
File Type	The type of file (example: Linear Hash).
Location	The operating system path name or drive designation for the medium on which the files for that volume are physically located. If this column is blank, the files are located on the default drive.
Volume Label	The label assigned to the volume (defaults to the time and date).

options

The (P*n*) option (enclosed within parentheses) sends output to the printer designated by the number *n*.

Correct Use of LISTVOLUMES Command

LISTVOLUMES
Displays the LISTVOLUMES popup on the screen.

LOCKS

The LOCKS command displays information about current locks for the workstation. Lock information is displayed for files that have transaction and domain control installed.

- ✎ For more information about transaction and domain control, see the chapter “Files and Records” in *Filing Systems* and the commands `CONTROL_ON` and `CONTROL_OFF`.

LOCKS

Using LOCKS

LOCKS displays a report of current locks. The report consists of:

- Names of files
- Keys
- Type of lock
- Mode of locking

- ✎ For information about locks and locking, see *Networking*.

Correct Use of LOCKS Command

LOCKS

A report of the currently locked files and records is displayed.

LOGOFF

Use LOGOFF to quit Advanced Revelation and return to the operating system.

OFF is a synonym for LOGOFF.

LOGOFF

Using LOGOFF

Use LOGOFF to terminate the present Advanced Revelation session and return to the operating system.

Correct Use of LOGOFF Command

LOGOFF

This command will terminate the Advanced Revelation session and return to the operating system.

LOGTO

The LOGTO command closes the current account and opens another. LOGTO moves the user to another account without leaving Advanced Revelation.

```
LOGTO account {,password}
```

Using LOGTO

LOGTO detaches the user from the current account. The user name does not change.

LOGTO detaches all files from the old account. When logging onto the new account, files on the default volume are attached.

account

Use any account name already defined in the SYSTEM file.

password

The password limits access to the account. Enter the correct password to access the account, if one has been assigned. If the account and password are correct, the program attaches the new account to the default drive and detaches any other drives.

If you do not include the account's password in the LOGTO sentence, the program displays the following prompt:

```
PASSWORD:
```

You will need to type in the correct password (which will not show on the screen) in order to access the account.

Correct Use of LOGTO Command

```
LOGTO AR, RECEIVE
```

The program moves to the account AR, which uses the password RECEIVE.

```
LOGTO TEST
```

The program moves to the account TEST.

```
LOGTO SYSPROG, A$2\\
```

The program moves to the account SYSPROG, which uses password A\$2\\.

MACRO

MACRO activates an existing macro set defined at the Macros window.

```
MACRO macro_name
```

Using MACRO

Use MACRO to run a macro set previously defined at the Macros window.

macro_name

The name of the macro set specified at the Macros window.

Correct Use of MACRO Command

```
MACRO BACKUP
```

The macro set BACKUP is activated.

MAKEACCOUNT

The MAKEACCOUNT command builds a new account.

CREATE-ACCOUNT is a synonym for MAKEACCOUNT.

MAKEACCOUNT account

Using MAKEACCOUNT

Use MAKEACCOUNT to create an entirely new account in Advanced Revelation. The new account will contain its own environment and vocabulary (VOC) files.

You must be in the SYSPROG account to use the MAKEACCOUNT command. MAKEACCOUNT writes a record in the SYSTEM file for the new account. The program creates a new VOC file and copies selected records from the SYSPROG VOC file to the new account's VOC file.

MAKEVOC looks to the SYSTEM file when creating a new account. The records NEW.ACCOUNT.VOC.LIST and NEW.ACCOUNT.DICT.VOC.LIST determine which vocabulary words and dictionary records will be copied to the new account. The SYSTEM record NEW.ACCOUNT.FILE.LIST determines which files will be made available to the new account.

There are two ways to log onto the new account. Log off and then log back onto the new account, or use LOGTO to move to the new account.

account

The *account* is the name to be used for the account.

The program prompts for a password, which is the character string required to log onto the account. The program then creates the account on the default volume.

Correct Use of MAKEACCOUNT Command

```
MAKEACCOUNT ACCTS.PAY
```

This statement creates a new account named ACCTS.PAY.

```
LOGTO ACCTS.PAY
```

This statement transfers system control to the new account.

WHO

This statement displays the account name.

MAKELIST

The R/LIST command MAKELIST creates and activates a list of record keys from data in a record or in a multivalued field.

FORM-LIST is a synonym for MAKELIST.

MAKELIST filename key {field} {option}

Using MAKELIST

MAKELIST converts all the fields in a record or all the values in a multivalued field into an active list of record keys.

Use MAKELIST to create an active list out of a relational index field or out of an inverted list. MAKELIST works somewhat like GETLIST but can be executed against a record in any file, and can also create lists out of multivalued fields.

filename

Use any attached Advanced Revelation file.

key

The *key* identifies which record in *filename* contains the data to make into a list of keys.

field

If *field* is specified, then MAKELIST will create an active list out of the multivalues in that field. If *field* is not specified, MAKELIST will create an active list out of all the fields in the record.

Field can be a field name or field number. If a field number is specified, MAKELIST will look in the dictionary of *filename* to determine which field to use in creating the active list.

option

Use the (S) option (enclosed within parentheses) to suppress message output.

Correct Use of MAKELIST Command

```
MAKELIST NAME.CREF SMITH 6
```

This command builds a list of record keys from the sixth field of the SMITH record in the NAME.CREF file.

MAKEUSER

MAKEUSER creates a new user name in the system and associates it with an account.

```
MAKEUSER {user} {account} {option}
```

Using MAKEUSER

Use MAKEUSER when you wish to create a new user profile in the system. Once the new user is created, the user name can be used to log into Advanced Revelation.

Each user is associated with a specific account. When a user logs into Advanced Revelation using the new user name the user will automatically enter into the account associated with the new user name, and be able to access files belonging to that account.

user

The *user* specifies a new name by which users can log into Advanced Revelation.

account

The *account* determines which set of files a user will have access to when logging in as the new user. The account must represent an existing account.

option

Use the (S) option enclosed within parentheses to suppress system messages.

Correct Use of MAKEUSER Command

```
MAKEUSER MIKE SYSPROG
```

This command creates a new user called MIKE and associates the new user with the SYSPROG account.

MAKEVOC

MAKEVOC creates an entry in the VOC file for an R/BASIC program.

CATALOG is a synonym for MAKEVOC.

MAKEVOC filename keys {options}

Using MAKEVOC

Use MAKEVOC to add the name of an R/BASIC program into the vocabulary (VOC) file. This will permit the program to be run from the TCL window by typing in the program name. (Compare this to the TCL command RUN.)

- ☛ All R/BASIC external subroutines must be cataloged with MAKEVOC in order for a main program to execute them. MAKEVOC will only catalog the object code of an R/BASIC program. It looks only for the "\$" version of the program and therefore does not require that the source code be available.

filename

Use *filename* to specify any attached Advanced Revelation file.

keys

Keys is made up of one or more record keys separated by spaces. Substitute an * (asterisk) as the record key to catalog all the records in a file. To use MAKEVOC on a selected list, *keys* must be empty.

MAKEVOC builds an entry in the vocabulary (VOC) file with the following format:

Field 1	RBASIC (the literal word R/BASIC)
Field 2	(null)
Field 3	BP (file name of the file containing the object code)
Field 4	CUST.ENTRY (object code name without "\$")

options

Options must be enclosed within parentheses. Three options are available:

Option	Meaning
(S)	Suppresses message output.
(B)	Catalogs the program so that it saves the current screen image and clears the screen before executing a main program (not SUBROUTINE). When the program finishes, the current screen image is restored.
(O)	Overwrites a VOC entry of the same name.

Correct Use of MAKEVOC Command

```
MAKEVOC BP TABLE.CHECK
```

This command will catalog the program \$TABLE.CHECK from the BP file.

```
MAKEVOC INV.BP STND.DEV COMP.INT
```

This command will catalog the programs \$STND.DEV and \$COMP.INT from the INV.BP file.

```
MAKEVOC INV.BP *
```

This will catalog all source code records from the INV.BP file.

```
SELECT BP WITH @ID MATCH "0X"
```

```
MAKEVOC BP (B)
```

The SELECT statement will select a list of records with record keys that begin with 0X. The MAKEVOC statement will catalog the three records from the BP file. When executed these records (programs) will save, clear, and restore the current screen.

MERGE

MERGE calls the Merge processor to combine data from an existing data file with a Merge template and output the resulting document.

```
MERGE template {key_list} {options}
```

Using MERGE

MERGE generates forms and reports by merging existing data with a previously defined template. The Merge template is defined at the MakeMerge window.

template

This is the name of the Merge template specified at the MakeMerge window.

key_list

This is an optional list of record keys identifying the data to merge. The data file will have been previously specified at the MakeMerge window.

options

These are output options for the merged document. The option(s) specified will override the corresponding specification at the Merge window. The available options are:

Option	Meaning
(P)	Output to printer. Character flags interpreted as printer codes.

Option	Meaning
(T)	Output to screen. Character flags interpreted as printer codes.
(F)	Output to output file specified at Merge template. Character flags and page breaks encoded, not interpreted.
(O)	Output to output device. Character flags and page breaks converted to printer codes.
(X)	Output to output device. Character flags and page breaks stripped.

Correct Use of MERGE Command

MERGE NEWFORM
The Merge template NEWFORM is executed.

MERGE NEWFORM SEPT
The Merge template NEWFORM is merged with data from the SEPT record.

MERGE NEWFORM SEPT (P)
The Merge template NEWFORM is merged with data from the SEPT record, and the resulting document is sent to the printer.

NAMEVOLUME

The NAMEVOLUME command initializes or renames a volume. It assigns a logical name to the volume and builds a volume directory.

NAMEMEDIA is a synonym for the NAMEVOLUME command.

```
NAMEVOLUME volume {identifier}
```

Using NAMEVOLUME

Use NAMEVOLUME to initialize a volume for use. NAMEVOLUME creates a REVVOL map on the volume.

NAMEVOLUME can also be used to rename the volume. You cannot rename a volume that contains files that have been indexed.

It is important that each volume created in Advanced Revelation have a unique name. This is especially important in environments where users may have access to network drives but can also attach volumes from a local drive. Failure to ensure uniqueness of volumes can result in incorrect updates of indexes.

- ✎ The `DEFINEFILE` command will initialize a volume automatically if the volume has not previously been initialized with `NAMEVOLUME`.

volume

Volume defines the volume for which you are establishing or renaming a directory. Use a colon as part of the drive specifications.

identifier

The *identifier* specifies the name to be given to the volume.

If *volume* is specified for an existing volume, the volume will be renamed (unless it contains indexed files). If no *volume* is specified for an existing volume, `NAMEVOLUME` displays the message:

The volume `VOLNAME` is currently named *identifier*

Correct Use of `NAMEVOLUME` Command

```
NAMEVOLUME B: INVOICE
```

This statement initialized the volume on B: by giving it the name INVOICE.

PAINT

The `PAINT` command invokes the paint processor, which is used to create, modify, and delete windows and forms.

- ✎ For information on using the window painter and the report painter, see “Creating Windows with Paint” in *Developing an Application*, and “Creating Forms” in *Creating Reports*.

```
PAINT      {filename} template
or
PAINT.FORM
```

Using `PAINT`

Use `PAINT` when you want to create, modify, or delete a window. Use `PAINT.FORM` when you want to create, modify, or delete a form.

filename

The *filename* can be any attached file name. The default *filename* is `TEMPLATES` for `PAINT` and `REPORTS` for `PAINT.FORM`. If no *filename* is entered, the template is read from the default file. Using *filename* allows you to store window and report templates in other files.

template

The *template* can be the name of a window or form template record.

Correct use of PAINT and PAINT.FORM

```
PAINT SALES.TEMP CUSTOMER
```

The *CUSTOMER* window template in the *SALES.TEMP* file is read by the window painter.

```
PAINT.FORM INVOICE
```

The *INVOICE* form in the *REPORTS* file is read by the form painter.

PASSWORD

The **PASSWORD** command allows you to change the password for the current account.

PASSWORD

Using PASSWORD

Use **PASSWORD** to create or modify the current account's password. **PASSWORD** restricts account access to authorized users.

The program prompts for a user password by displaying:

```
Enter Old Password?
```

```
Enter New Password?
```

☞ Passwords will not appear on the screen as they are being typed.

The prompt *Enter Old Password?* appears only if there is an existing password for the account.

At *Enter New Password?*, type the new password.

Passwords can be as long as desired and can contain any character, including spaces.

Correct Use of PASSWORD Command

```
PASSWORD
```

```
Enter Old Password? ZUMZUM
```

```
Enter New Password? XYPHIS
```

The password for the *PAYROLL* account changes from *ZUMZUM* to *XYPHIS*. The message "Changed" will appear after **PASSWORD** has successfully changed the password.

PC

The PC command allows you to enter the operating system environment or call any operating system command from within Advanced Revelation.

```
PC {EXIT} {oscommand}
```

Using PC

Use PC to execute external programs (such as spreadsheets, word processors, utilities, etc.) from within Advanced Revelation. Unlike SUSPEND, PC does not roll Advanced Revelation out of memory; Advanced Revelation will continue to occupy about 200K of RAM.

☞ In OS/2 versions, PC and SUSPEND are identical in operation.

The PC command must have access to the operating system command processor. Under MS-DOS or OS/2, this may be accomplished indirectly by using the SET COMSPEC command. See the SUSPEND command for additional information.

You can change the default drive while in the operating system environment, but Advanced Revelation resets the default to its initial value when you exit to Advanced Revelation. To return to Advanced Revelation from the operating system, type EXIT at the operating system prompt. See your operating system manual for further command information.

☞ Do not execute any memory-resident utilities while Advanced Revelation is suspended.

EXIT

Preceding *oscommand* with the EXIT option returns control directly to Advanced Revelation without prompting the user to press [Enter] first.

oscommand

Use *oscommand* to perform any specific operating system operations. For example:

```
PC DIR B:*.*COM
```

When the DIR operation is complete, you are prompted to press [Enter] to return to Advanced Revelation.

Correct Uses of the PC Command

```
PC
```

```
The IBM Personal Computer DOS
```

```
Version 3.1 (C) Copyright IBM Corp 1981, 1982, 1983, 1984
```

```
C:>chkdsk B:
```

```
362496 bytes total disk space
 37888 bytes in 2 hidden files
134144 bytes in 50 user files
190464 bytes available on disk
```

```
654336 bytes total memory
214424 bytes free
```

```
C:>EXIT
```

The PC command exits Advanced Revelation, enters MS-DOS, and displays the C:> prompt, signaling its readiness for an MS-DOS command. MS-DOS processes the CHKDSK command on disk drive B:, completes the operation, and again displays the C:> prompt. Typing the MS-DOS EXIT command returns the Advanced Revelation program to its initial default drive and program location.

PDISK

PDISK routes printer output to an operating system file or back to the printer.

```
PDISK [os_file] {options}
      [PRN]
```

Using PDISK

Use PDISK to cause subsequent Advanced Revelation printer output (as specified by the (P) or LPTR options) to be printed to an operating system file.

PDISK is useful when you wish to “spool” a report or program output to disk. This permits you to generate and “print” a report without needing a printer available. Once the report or program has finished spooling to disk, it can be printed to a printer, copied to another machine, etc.

Data will be written to the specified operating system file exactly as if it were being printed. The output will be in ASCII format (lines delimited with CR/LF, file terminated with ASCII 26). If appropriate, top-of-form characters (ASCII 12) will be embedded into the output text. All formatting options such as headers, footers, and column headings are printed to file.

Printer output will be routed to the file indicated by *os_file*. To route the output of different reports or programs to different files, repeat the PDISK command before each program, indicating a new operating system file name.

You must execute PDISK before and after your report. The first PDISK opens the file for printer output; the second is required to close the file.

os_file

Use *os_file* to indicate the name of an operating system file to which you wish to route output. If the file already exists, you will be prompted to overwrite it. If printer output is already redirected to another file, the original output file will be closed and a new file opened.

PRN

Once printer output has finished, the command

```
PDISK PRN
```

will close the operating system spool file and reroute printer output back to the printer.

options

Options must be enclosed within parentheses. Two options are available with the *PDISK os_file* statement:

Option	Meaning
(O)	Overwrites any existing <i>os_file</i>
(S)	Suppresses system messages

Correct Use of PDISK Command

```
PDISK C:\CUSTOMER.RPT
```

All subsequent printer output is routed to the operating system file C:\CUSTOMER.RPT.

```
PDISK PRN
```

The current printer output file (if any) is closed, and printer output is redirected back to the printer.

PORT

The **PORT** command displays the current communication settings.

```
PORT
```

Correct Use of PORT

```
PORT
```

The current communications settings are displayed.

QUERY

The R/LIST command QUERY displays a data entry window in Query mode.

QUERY window

Using QUERY

Use QUERY to display a data entry window with Query mode active.

window

The name of a data entry window. The window template must exist in the TEMPLATES file.

Correct Use of QUERY Command

QUERY NEW.CUSTOMERS

The data entry window NEW.CUSTOMERS is displayed with Query mode active.

RECORDCOPY

The RECORDCOPY command copies individual records within a file, from one file to another, or to the screen or printer. See the FILECOPY command to copy entire files.

**RECORDCOPY {DICT} file keys {TO:} {target_DICT} ⇨
{target_file} {target_keys} {options}**

Using RECORDCOPY

Use RECORDCOPY to copy individual Advanced Revelation records to new locations or to display them on the screen or printer.

RECORDCOPY can copy records to:

- The same file under a different record key
 - A different file (with an option of changing the record key during the copy)
 - The screen
 - The printer
 - From, to, or between operating system files.
- ☛ Advanced Revelation considers operating system files to be records in the special file DOS. Thus a single operating system file copied into Advanced Revelation will produce one record. Each Advanced Revelation record copied to the file DOS will produce one operating system file. Copy only those operating system files with less than 65,535 characters.

You can enter the full command, or be prompted for the parameters. If you press [F9] or [Enter] after entering the command, a window will prompt you for any remaining parameters.

DICT

If DICT is specified, records will be copied to or from the dictionary of the *file*.

file

File indicates the name of the file that contains the records to be copied. If copying from an operating system file, use the special filename DOS.

keys

Use *keys* to specify the record(s) to be copied. The *keys* parameter is made up of one or more record keys separated by spaces. Substitute * (asterisk) as the key to copy all the records in a file.

To use RECORDCOPY to copy a selected list of record keys, the source key must be empty.

If copying from operating system files, be sure to include for each source key the full path and operating system filename, including the extension (for example, C:\DATA\INVOICE.PGM). Remember that each operating system file will become one record in an Advanced Revelation file.

target_DICT

If *target_DICT* is specified, records will be copied to the dictionary of *target_file*.

target_file

Target_file indicates the name of the Advanced Revelation file into which the record(s) will be copied.

- ☛ When copying to another file, *target_file* must be preceded by an open parenthesis.

If copying to an operating system file, use the special filename DOS.

target_keys

If provided, *target_keys* will be used to rename the record(s) as they are copied to *target_file*. Records are renamed using a one-to-one correspondence with the source keys.

If there are more source keys than target keys, the remaining source keys will not be changed as the copy progresses.

When copying to operating system file(s), be sure to include for each destination key the full path and operating system filename, including the extension (for example, C:\DATA\INVOICE.PGM). Remember that each record in an Advanced Revelation file will become one operating system file.

options

Options must be enclosed within parentheses.

If no options are specified, a null input at the TO: prompt window will cause RECORDCOPY to copy the records to the screen.

Use the options together to achieve the results you want, within the restrictions noted below.

Option	Meaning
(A)	Prints a message for each record copied
(C)	Suppresses a top-of-form for each record sent to the printer or terminal
(D)	Deletes the source record(s) after successfully copying to the target record(s). D has no effect on the P or T options.
(E)	Suppresses error message output
(I)	Converts Advanced Revelation data into the proper operating system file format. Field marks (ASCII 254) are changed to carriage return/line feeds. The record(s) are terminated with a [Ctrl-Z] (ASCII 26).
(K)	Eliminates line number output
(L)	Locks the records in the source file and the target file as they are being copied
(N)	Overwrites existing target record(s) only when there are corresponding source and target records. If you try to copy a record that does not exist, an error message is displayed.
(O)	Overwrites any target records (has no effect on the P or T options)
(P)	Copies record(s) to the printer. There is no TO: prompt.
(R)	Converts operating system file data into the proper Advanced Revelation format. Carriage return/line feeds are changed to field marks (ASCII 254) and [Ctrl-Z] termination marks are removed.

Option	Meaning
(S)	Suppresses message output
(T)	Copies record(s) to the terminal. There is no TO: prompt.
(X)	Copies the source record(s) in hex format

Correct Use of RECORDCOPY Command

RECORDCOPY ORDERS 123 456 TO: 777 888
Copies the record 123 in the ORDERS file to record 777 and record 456 to record 888.

SELECT ORDERS WITH INV.DT LT FRIDAY
RECORDCOPY ORDERS TO: (NEWORDERS)
This sequence of commands selects records from ORDERS and copies them to the NEWORDERS file.

RECORDCOPY DICT ORDERS * TO: (DICT NEWORDERS)
This command copies all the records in the ORDERS dictionary to the dictionary of NEWORDERS.

RECORDCOPY ORDERS 123 (T)
This command copies record 123 to the terminal.

RECORDCOPY ORDERS 123 (OID) TO: (DOS A:INVOICE.123)
This copies record 123 on the Advanced Revelation file ORDERS over DOS file INVOICE.123 in operating system format on drive A. The command deletes record 123 when the copy is complete.

RECORDCOPY DOS A:INV.DAT TO: B:INV.DAT
This command copies DOS file INV.DAT on drive A to DOS file INV.DAT. (Assume INV.DAT does not exist on drive B.)

REDEFINEFILE

The REDEFINEFILE command changes the file management system or recalculates the modulo of a file.

```
REDEFINEFILE [{DICT}] filename {TO:} {vol} {attributes} ⇨  
              [{DATA}]  
  
              {FIELDS = [field_def ::]} {option}
```

Using REDEFINEFILE

Use REDEFINEFILE to redefine or resize a file.

- ☛ Advanced Revelation can use, but cannot create, files using the LINK format used in previous versions of Advanced Revelation.

REDEFINEFILE works by creating a new file according to the specifications given in the command. It then copies all records from the old file to the new file. **REDEFINEFILE** then deletes the old file and renames the new file to match the old file name.

- ☛ If the file being recreated has been indexed, all indexes must be removed from the file before recreating it. Once the file has been recreated, the indexes can once again be established.

vol

Vol (volume) defines the complete directory path, including the drive specifications. Use a colon as part of the drive specification. *Vol* can also be the name of a volume record in the VOC file. See **SETVOLUME**.

When a file is being converted to a new filing system, *vol* will determine the new filing system.

DICT

Use **DICT** to specify that only the dictionary file be recreated.

DATA

Use **DATA** to specify that only the data file be recreated.

- ☛ If you do not specify either **DICT** or **DATA**, the program recreates both the dictionary and the data files.

filename

Use *filename* to specify the name of the attached file to be remade. Do not use a Qfile as the file name.

attributes

The *attributes* you may specify vary according to the filing system or bonding. Refer to the “Files and Records” chapter of *Filing Systems* or to the bond documentation for the appropriate specifications.

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of REDEFINEFILE Command

```
REDEFINEFILE AP 500 50 15
```

This recreates a file called AP on the current default volume using the new parameters 500 50 15 (which are specific to the current file type).

RENAMEFILE

RENAMEFILE changes the name of a file.

```
RENAMEFILE [ {DICT} ] file [ {new_account} ] new_file {option}
           [ {DATA} ]      [ {GLOBAL} ]
```

Using RENAMEFILE

Use RENAMEFILE to assign a new Advanced Revelation file name to an existing file. RENAMEFILE will update the appropriate volume directory. Any indexing associated with a file must be removed before you can rename the file.

- ✎ If you are using a network system, you must re-attach the renamed file, because the program changes the volume directory.

DICT

Use DICT before the file name to rename only the dictionary portion of the file.

DATA

Use DATA before the filename to rename only the data portion of the file.

- ✎ If you do not specify either DICT or DATA, the program renames both the dictionary and the data files.

file

Use *file* to specify the file to be renamed. The file cannot be a Qfile.

new_account

If *new_account* is specified, the file will be attached to the account specified in *new_account*. The new account must be an existing account.

GLOBAL

Using the keyword GLOBAL, you can make a file accessible from any account.

Any indexes associated with the file must first be removed. The indexes can then be reinstalled after the RENAMEFILE process.

- ✎ You must attach the file (using the ATTACH command or window) before attempting to install or reinstall indexes for a global file.

new_file

Use *new_file* to specify the new name of the file. When you specify a valid *new_filename*, the program updates the FILES file.

option

Use the (S) option (enclosed in parentheses) to suppress message output.

Correct Use of RENAMEFILE Command

RENAMEFILE NOTES DIARY

Changes the name of the Advanced Revelation file NOTES to DIARY.

RESET

RESET does a generalized system reset, re-initializing your current Advanced Revelation session.

RESET

Using RESET

On occasion, a problem in the system (such as a program that has aborted) will cause the window display or the value of system variables to be incorrect. Executing RESET will reinitialize system variables and repaint the window backdrop and status line. Executing RESET is the equivalent of executing an ABORT ALL from within R/BASIC.

RESET does the following:

- Reloads the environment and color settings for the current user or account
- Resets system variables
- Reinitializes and repaints the status line
- Repaints the backdrop
- Loads the default macro key set

If RESET is executed from a TCL level other than 1, it reinitializes the current window and repaints the status line.

Correct Use of RESET Command

RESET

Advanced Revelation reinitializes system variables and reloads environment and video settings. The background and status line are repainted.

RETURN

The RETURN command returns control one TCL level closer to level one.

RETURN

Using RETURN

Advanced Revelation allows users to interrupt an R/BASIC program or the debugger and execute TCL commands, or to call TCL directly. All variables of the program being interrupted are preserved. Up to eight recursive levels of TCL can be called.

RETURN is used at TCL to terminate that level of TCL and resume the program from which TCL was called.

The WHO command will report the current recursive level of TCL.

Correct Use of RETURN Command

Line 24 'BACKUP' broke because the BREAK key was hit.

```
! EXECUTE RUN BP STAT
```

Line 13 'STAT' broke because the BREAK key was hit.

```
! END
```

```
: RETURN
```

```
E Line 24 'BACKUP'  
!
```

This shows the Debugger interrupting the BACKUP program. EXECUTE sends Advanced Revelation to the program STAT. When STAT is complete, END is typed, and RETURN sends Advanced Revelation back to the initial Debugger program BACKUP. Notice that the program returns to line 24.

RLIST

The RLIST command executes an R/LIST statement template created earlier using the Rlist window.

```
RLIST template.file template.record { (options) }
```

template.file For *template.file*, enter the name of the file in which your predefined LIST statement, or template, is stored. The default RLIST template file is **QUERIES**.

template.record The *template.record* is the name of the record in the template file that you want to execute.

If you have activated a filter (select list) before executing the RLIST command, only the records in the select list will be displayed.

options If you specify no options, your report will be displayed to the terminal. If you specify other options, they must be enclosed in parentheses. Other options are:

- P Sends the report to the printer.
- E Suppress the initial form feed.
- N “No page” — do not stop at the end of each page (not applicable for reports sent to a printer).
- X Capture R/LIST source code in the operating system file RLIST.
- SX Capture R/LIST source code and do not execute the report.

Correct Use of RLIST Command

```
RLIST QUERIES PHONE.LIST (P)
```

This statement executes the PHONE.LIST report from the QUERIES file and directs the output to a printer.

ROLLOUT

The ROLLOUT command specifies the name for the operating system rollout file needed by the SUSPEND command to write Advanced Revelation to an operating system file.

```
ROLLOUT filename
```

Using ROLLOUT

To store Advanced Revelation to an operating system file, you must first either set up a rollout file using the Environment window, or use ROLLOUT and specify a filename. The system variable @ROLLOUT.FILE will contain the current operating system filename to be used by SUSPEND.

This sets up the conditions required to use SUSPEND. If you use SUSPEND when the @ROLLOUT.FILE is empty, the program acts as though you used the TCL commands PC.

filename

Use *filename* to specify any valid operating system file name. Include a path name if the ROLLOUT file is not to be on the default drive.

Refer to the SUSPEND statement in the “R/BASIC Command Reference” chapter of *R/BASIC*.

Correct Use of ROLLOUT Command

```
ROLLOUT REV.SUS
```

This command sets up REV.SUS on the default drive as the operating system filename for SUSPEND.

RUN

The RUN command executes R/BASIC programs that have been previously compiled using the COMPILE command.

```
RUN filename key {options}
```

filename

Use *filename* to define any attached Advanced Revelation file that contains R/BASIC object code.

key

Use *key* to identify an R/BASIC program that has been compiled using COMPILE. RUN searches for the \$ (dollar sign) appended to the front of the key the signifies R/BASIC object records.

- ☞ All R/BASIC external subroutines must be cataloged (have pointers in written to the VOC file using MAKEVOC).

options

Options must be enclosed within parentheses. More than one option can be specified at one time. Three options are available.

Option	Meaning
(B)	The B option causes Advanced Revelation to save the current screen image and blank the screen before executing the program. When the program finishes, the current screen image is restored.
(D)	Use D to enter the Debugger immediately after executing a program. See DEBUG in the “R/BASIC Command Reference” of R/BASIC.
(P)	Use P to output to a printer.

Correct Use of RUN Command

- RUN BP PGM123
This command statement executes program \$PGM123 in the BP file.
- RUN BP PGM123 (P)
This command statement executes \$PGM123 and sends output to the printer.
- RUN BP PGM123 (D)
This command statement sends Advanced Revelation to the Debugger immediately after beginning execution of program \$PGM123.

RUNMENU

Use RUNMENU to display menus from TCL and TCL batch records. RUNMENU is commonly used in logon verbs to start the first menu of an application.

```
RUNMENU {menu}
```

menu

Use *menu* to specify the menu to display. The default menu is MAIN, the main system menu. The menu name is a menu template in the MENUS file.

Correct Use of RUNMENU

```
RUNMENU SALES.MASTER
```

The SALES.MASTER menu is displayed.

SAVELIST

The R/LIST command SAVELIST saves a list of the record keys to the LISTS file after the list has been created using MAKELIST, SELECT, or SSELECT.

```
SAVELIST {listname} {option}
```

Using SAVELIST

Use SAVELIST to permanently save a list of record keys created with SELECT, SSELECT, or MAKELIST. SAVELIST saves the list of record keys in the LISTS file. Each field in the list record contains one record key.

Lists of records saved using SAVELIST can be edited or processed repeatedly, without restrictions. This is useful if you have selected records using a complex set of criteria, or use the same records in several different reports.

Retrieve lists using GETLIST, and when they are no longer needed, delete using DELLIST. If information in the list needs changing, use EDIT.

listname

Use *listname* to define the name of the record to be stored in LISTS. If *listname* is undefined, SAVELIST assigns "" (null) as the list name.

The program divides lists of keys with more than 32,767 characters into multiple lists, saving the first segment under the *listname*. Additional segments receive numbers concatenated to the end of *listname* with an "*" (asterisk). For example:

```
END.OF.YEAR listname
END.OF.YEAR*2 second segment of listname
```

The segment number is transparent when used by GETLIST but is required when accessing a list with another process, such as EDIT or an R/BASIC program.

option

Use the (S) option (enclosed within parentheses) to suppress message output.

Correct Use of SAVELIST Command

```
SSELECT 6 CUSTOMERS
SAVELIST SAMPLE
```

Six CUSTOMER file records are sort selected and saved in the LIST file as SAMPLE.

SELECT

The R/LIST command **SELECT** creates and activates a list of record keys that meet certain criteria. The list of active keys is then available to the next Advanced Revelation command.

```
SELECT {n} {DICT} filename {USING alt_filename} ⇨  
  {selection_criteria} {sort_criteria} {keywords} {option}
```

Using SELECT

SELECT assembles a group of records from a file and stores their keys temporarily in a list of record keys. The program offers these selected records to other commands for processing.

When the program selects a group of records, it stores only the record keys in the list of record keys. Other commands, like **LIST** or **RECORDCOPY**, access the data from this list of record keys.

Once the program selects and orders the records, the status line will indicate that a filter is active. At this point the next command can access the list.

Ordinarily, **LIST** reports all the records in the **CUSTOMER** file. In this case, “filter” on the status line indicates that a list of record keys is active. Thus, **LIST** reports only those five records previously selected. Once an active list has been used it is cleared.

To use a list more than once, save and retrieve it by using the **SAVELIST** and **GETLIST** commands. Use the **CLEARLIST** command to deactivate an unwanted list.

For more information about active filters, see the *Using Advanced Revelation* section, and “Using Filters” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

n

Use *n* to specify the number of records to be selected from *filename*. This number designates a random selection of the first *n* records found on file.

DICT

Use **DICT** to specify that only the dictionary records be selected from *filename*.

filename

Use *filename* to specify the name of any attached Advanced Revelation file.

USING

A **USING** clause specifies that the **SELECT** operation is to use the dictionary of an alternative file, rather than the *filename* dictionary.

alt_filename The name of the alternate file (not the dictionary name)

See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria enables you to indicate a subset of records to process by including filtering specifications. For more information about selection criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

sort_criteria

Sort_criteria enables you to specify the order by which records are to be displayed. For details about sort criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

keywords

The *R/LIST keywords* include flow of control, formatting, and output specifications. For details on the use of these keywords, see the “R/LIST Keyword Reference” chapter of the *R/LIST* section.

option

Use the (S) option enclosed in parentheses to suppress message output.

Correct Use of SELECT Command

```
SELECT PARTS WITH DESC CONTAINING "MOTOR" BY PART.NO
RUN PGMPART.ENTRY
```

SELECT finds the records in the PARTS file that contain the word 'MOTOR' in the description. The R/BASIC program PGMPART.ENTRY processes only these records.

```
SELECT DICT CUSTOMERS WITH FMC BETWEEN 5 AND 10
SELECT DICT CUSTOMERS WITHOUT F1 "S" BY FMC
LIST DICT CUSTOMERS DICT.INFO
```

SELECT first finds the records in the dictionary of the CUSTOMERS file in which the field number is between 5 and 10 (inclusive). SELECT is once again invoked to further refine the list of records by selecting fields without 'S' in field one and sorting the list by field number. Finally, the records are listed to show the field DICT.INFO.

SET-CRT

The SET-CRT command defines the CRT screen width and height dimensions for certain Advanced Revelation processes.

If no width or height parameters are specified, SET-CRT will report the current settings.

SET-CRT {width {,height}}

Using SET-CRT

The default for the monitor display is 80 columns across, 22 lines down. Use SET-CRT to modify these default settings.

The settings established for screen size are used by windows and popups in Advanced Revelation. In addition, page-oriented TCL commands (BLIST, RECORDCOPY) use the settings to size their output.

R/LIST verbs (LIST and SORT) will use the screen height to determine the size of the screen page. The R/BASIC statements HEADING and FOOTING will use the screen height established by SET-CRT.

width

Use *width* to set the number of display columns across the screen.

height

Use *height* (preceded by a comma) to set the number of display rows down the screen. To specify *height* without a *width* parameter, precede *height* with a space and comma.

☛ To access the width and height settings in R/BASIC, use the @CRTHIGH and @CRTWIDE functions.

Correct Use of SET-CRT Command

SET-CRT

This command displays the existing settings.

SET-CRT 40

This command sets the CRT column width to 40, while the height remains at the default setting of 22.

SET-LPTR

The SET-LPTR command defines the printer page's width and height dimensions for certain Advanced Revelation processes.

If no width or height parameters are specified, SET-LPTR will report the current settings.

```
SET-LPTR {width {,height}}
```

Using SET-LPTR

The default for the printer is 132 columns across, 63 lines down. Use SET-LPTR to modify these default settings.

The settings established for printer size are used by page-oriented TCL commands (BLIST, RECORDCOPY). R/LIST verbs (LIST and SORT) will use the printer height to determine the size of the page. The R/BASIC statements HEADING and FOOTING will use the printer page height established by SET-LPTR.

width

Use *width* to set the number of columns across the page.

height

Use *height* (preceded by a comma) to set the number of lines down the page. To specify *height* without a *width* parameter, precede *height* with a space and comma.

- ☛ To access the width and height settings in R/BASIC, use the @LPTRWIDE and @LPTRHIGH functions.

Correct Use of SET-LPTR Command

```
SET-LPTR 80,60
```

This sets the printer page configuration to 80 characters across and 60 lines down.

```
SET-LPTR ,40
```

This sets the printer page configuration to 40 lines down, while the characters across remains at 132.

SET-OPTIONS

The SET-OPTIONS command allows you to display the current system default settings and change them and to change Advanced Revelation's mode of operation.

```
SET-OPTIONS {options} {ext_options}
```

Using SET-OPTIONS

Use SET-OPTIONS to change Advanced Revelation's mode of operation and the various system options such as printer output and message suppression. Changes made with SET-OPTIONS remain in effect until logoff.

options

Options may be any of the standard system options or an empty set of parentheses. The empty parentheses resets any option settings to the system default settings.

Use SET-OPTIONS with a standard option to change any of the system default settings. Once an option has been set with SET-OPTIONS, all subsequent TCL commands will act as if that option had been included as part of the command.

Using SET-OPTIONS with no options will return the status of the current settings.

Correct Use of SET-OPTIONS Command

```
SET-OPTIONS (P)
```

Any TCL command that has the P option available will now automatically send output to the printer.

```
SET-OPTIONS
```

Any option settings are displayed.

```
SET-OPTIONS ()
```

Resets any options set to the system default settings.

SETFILE

The SETFILE command builds a pointer to a file in the same or another account.

```
SETFILE {vol} account {,password} [{DICT}] filename ⇨
      {Qfile} {options}
```


Using SETFILE

Use SETFILE to create a synonym for a file in the current account or to get access to a file on another account.

Synonyms for files in the current account are useful if the original file name is unwieldy. They are also useful if you need access to a file that cannot be attached because a file with a duplicate name is already attached. SETFILE also enables you to create a name in the current account to access a file belonging to another account. This prevents you from having to duplicate the data in the current account.

The synonym you create is stored in the FILES file. You may thereafter refer to the original file by the synonym you created.

SETFILE creates temporary synonyms. They are valid for as long as you are logged on; once you log off, the synonymous name is lost. To create permanent synonym names, use the W option. Permanent synonyms are stored in the QFILES file; if no QFILES file exists, they are stored in the VOC file.

vol

Use *vol* (volume) to specify the volume for the file.

account

Specify the *account* if you are creating a synonym for a file in another account.

password

Use *password* (preceded by a comma) when access to the account specified by *account* requires a password.

DICT

Use DICT to specify that the Qfile pointer point only to the dictionary portion of the file specified by *filename*.

DATA

Use DATA to specify that the QFILE pointer point only to the data portion of the file specified by *filename*.

☞ If neither DICT nor DATA are specified, both portions of the file are used.

filename

Use *filename* to specify the name of the file for which you want to create a synonym.

Qfile

Use *Qfile* to define the synonym you want to use to point to the file specified by *filename*. When *Qfile* is not specified, the system identifier QFILE becomes the default synonym.

options

Options must be enclosed within parentheses. Two options are available.

Option	Meaning
(S)	Suppress message output.
(W)	Store the synonym permanently.

Correct Use of SETFILE Command

```
SETFILE B: GENERAL.LEDGER CHART
```

This command creates a synonym named QFILE to point to the file named CHART in the GENERAL.LEDGER account that resides on volume B.

SETVOLUME

SETVOLUME creates a logical name for a volume on which Advanced Revelation files can be stored.

For more information about volumes, see the “Advanced Revelation Volumes” chapter of *Filing Systems*.

```
SETVOLUME vol file_sys location control_files_loc
```

Using SETVOLUME

Use SETVOLUME to create a name for an addressable volume, such as an operating system path or drive name. All commands which require a volume or drive identifier (for example, ATTACH, LISTVOL) can use the volume name established with SETVOLUME.

SETVOLUME creates a VOLUME record in the VOC file of the current account. The layout of the VOC entry is:

Field	Data	Description
001	VOLUME	The literal word VOLUME.
002	file type.	The name of a program used by the system to “decode” the files on the volume).
003	os_path	The operating system pathname for the medium on which the files are located.

For example, a volume called INV.DATA might be created, which could have a layout like this:

Field	Data
001	VOLUME
002	RTP57
003	C:\REV\INV.DATA

Once created, the volume could be attached by using the command ATTACH INV.DATA.

- More than one volume name can point to the same subdirectory. Each volume name can reference a different filing structure.

vol

A valid identifier for a volume.

file_sys

The name of the filing system which is used to do file I/O operations for the volume.

location

The operating system path and drive identifier for the files.

control_files_loc

The operating system path and drive identifier for the location of the Advanced Revelation system files maintained for bonded files.

Correct Use of SETVOLUME Command

```
SETVOLUME INVOICES RTP57 C:\REV\INVOICES
```

This example creates a volume pointer in the VOC file called INVOICES with the following layout:

```
field1 VOLUME
field2 RTP57
field3 C:\REV\INVOICES
```

All Advanced Revelation LHASH files on the C:\REV\INVOICES subdirectory can be accessed via the volume name INVOICES.

SORT

The SORT command creates a columnar report out of data in an Advanced Revelation file. SORT includes options for selecting, sorting, and displaying records in the file. After completing any *sort_criteria* sorts (see below), SORT will additionally sort the report by the key field of the file.

```
SORT {n} {ONLY} {DICT} filename {USING alt_filename} ⇨
      {selection_criteria} {sort_criteria} {fields} ⇨
      {keywords} {options}
```

Using SORT

Use SORT to produce formatted output from an Advanced Revelation file. SORT creates reports in which each record is printed as a row; fields from the record are printed in columns.

SORT differs from the LIST command only in that it always adds one additional sort level. SORT always sorts the key field of the file (@ID) as the last-level sort field. These two commands will have the same effect:

```
LIST VOC BY @ID
```

```
SORT VOC
```

n

Use *n* to specify the number of records to sort. The result will be a random selection of the first *n* records in the file.

ONLY

Use ONLY to use only record keys in the report or form. It takes no selection or sort criteria.

DICT

Use DICT to sort only the dictionary portion of the file.

filename

Use *filename* to specify any attached Advanced Revelation file.

USING

A USING clause specifies that the report or form is to use the dictionary of an alternative file, rather than the *filename* dictionary.

<i>alt_filename</i>	The name of the alternate file (not the dictionary name)
---------------------	--

See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria enables you to indicate a subset of records to display by including filtering specifications in the SORT command.

For more information about selection criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

sort_criteria

Sort_criteria specifies the order in which records are to be displayed.

For details about sort criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

fields

Fields designates the fields that are to be displayed for each record. The fields should be specified in the order in which they are to be displayed across the page, left to right.

If no fields are specified, SORT will use the list of field names kept in the dictionary records @CRT (for screen output) or @LPTR (for printer or disk output). If neither of these dictionary records exists in the dictionary of *filename*, SORT will use the field names defined in the @CRT or @LPTR records in the dictionary of the VOC file.

keywords

The *R/LIST keywords* include flow of control, formatting, and output specifications.

For details on the use of these keywords, see the “R/LIST Keyword Reference” chapter of the *R/LIST* section.

options

The following options are available and must be enclosed within parentheses.

Option	Meaning
(E)	Use E to suppress the initial top-of-form when printing to the printer.
(N)	Normally, when the output goes to the screen, the program waits at the end of each page. Use N if you want the report to scroll continuously.
(P)	Use P to output the printer.
(X)	Use X to cause SORT to save generated R/BASIC code to the operating system file RLIST and execute the report.
(SX)	Use SX to cause SORT to save generated R/BASIC code to the operating system file RLIST but not execute the report.

Correct Use of SORT Command

```
SORT CUSTOMERS
```

This will produce a sorted report of customers and display it on the @ID field of the CUSTOMERS file.

```
SORT CUSTOMERS (P)
```

This will produce the customer report sorted by customer number and printed to the printer.

```
SORT ORDERS IF INV.AMT GE "100" BY ORDER.NO INV.DT TOTAL INV.AMT
```

This will produce a report of orders with invoice amounts greater than or equal to \$100, sorted by order number. The report will total the invoice amount field.

SQL

The SQL command displays the SQL window, or executes the specified SQL statement.

```
SQL {statement}
```

statement

The *statement* may be any valid SQL statement. Refer to the “SQL Reference” chapter of *SQL*.

Correct Use of SQL Command

```
SQL
```

Displays the SQL window

```
SQL CREATE TABLE CUSTOMERS (CUST_NO INTEGER KEY, FNAME CHAR(12),  
LNAME CHAR(25), ADDRESS VARCHAR, CITY CHAR(30), ST CHAR(3))
```

The CUSTOMERS table is created with six columns defined.

SSELECT

The R/LIST command SSELECT creates and activates a list of record keys that meet certain criteria. The list of active keys is then available to the next Advanced Revelation command. After completing any sort criteria specified, SSELECT will additionally sort the report by the key field of the file.

```
SSELECT {n} {DICT} filename {USING alt_filename} ⇨  
  {selection_criteria} {sort_criteria} ⇨  
  {keywords} {option}
```

Using SSELECT

SSELECT differs from the SELECT command only in that it always adds one additional sort level. SSELECT always sorts the key fields of the file (@ID) as the last-level sort field. These two commands will have the same effect:

```
SELECT VOC BY @ID
```

```
SSELECT VOC
```

SSELECT assembles a group of records from a file and stores them temporarily in a list of record keys. The program offers these selected records to other commands for processing.

When the program selects a group of records, it stores only the record keys in the list of record keys. Other commands, like LIST or RECORDCOPY, access the data from this list of record keys.

Once the program selects and orders the records, the status line will indicate that a filter is active. At this point the next command can access the list. For example:

```
SSELECT CUSTOMERS WITH ST = 'WA'  
LIST CUSTOMERS
```

Ordinarily, LIST reports all the records in the CUSTOMERS file. In this case, “filter” on the status line indicates that a list of record keys is active. Thus, LIST reports only those five records previously selected. Once an active list has been used it is cleared.

To use a list more than once, save and retrieve it by using the SAVEDLIST and GETLIST commands. Use the CLEARLIST command to eliminate an unwanted list.

For more information about active filters, see the “Introduction to R/LIST” chapter of *R/LIST*.

n

Use *n* to specify the number of records to be selected from *filename*. This number designates a random selection of the first *n* records found on file.

DICT

Use DICT to specify that only the dictionary records be selected from *filename*.

filename

Use *filename* to specify the name of any attached Advanced Revelation file.

USING

A USING clause specifies that the SSELECT operation is to use the dictionary of an alternative file, rather than the *filename* dictionary.

alt_filename The name of the alternate file (not the dictionary name)

See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria enable you to indicate a subset of records to process by including filtering specifications.

For more information about selection criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

sort_criteria

Sort_criteria enable you to specify the order by which records are to be displayed.

For details about sort criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

keywords

The R/LIST *keywords* include flow of control, formatting, and output specifications.

For details on the use of these keywords, see the “R/LIST Keyword Reference” chapter of the *R/LIST* section.

option

Use the (S) option enclosed in parentheses to suppress message output.

Correct Use of SSELECT Command

```
SSELECT PARTS WITH DESC CONTAINING "MOTOR" BY PART.NO
RUN PGMPART.ENTRY
```

SSELECT finds the records in the PARTS file that contain the word 'MOTOR' in the description and sorts the list by part number. The R/BASIC program PGMPART.ENTRY processes only these records.

SUM

The R/LIST command SUM totals one specified field for selected records in a file.

```
SUM {DICT} filename {USING alt_filename} {selection_criteria} ␣
    {keys} {field} {options}
```


Using SUM

Use SUM to generate a total for any numeric field specified in the command.

n

Use *n* to specify the number of records to sort. The result will be a random selection of the first *n* records in the file.

DICT

Use DICT to perform the SUM operation based only on the dictionary of the file.

filename

Use *filename* to specify any attached Advanced Revelation file.

USING

A USING clause specifies that the report or form is to use the dictionary of the alternative file, rather than the *filename* dictionary.

alt_filename The name of the alternate file (not the dictionary name)

See the “R/LIST Keyword Reference” chapter of the *R/LIST* section for more information.

selection_criteria

Selection_criteria enable you to indicate a subset of records to display by including filtering specifications in the SUM command.

For more information about selection criteria, see “Using Filtering” in the “Introduction to R/LIST” chapter of the *R/LIST* section.

keys

Keys is a list made up of one or more record keys separated by spaces.

Use *keys* to specify the records containing the *field* you want to total. To perform the SUM operation on all of the records in a file, *keys* can be left empty.

field

Use *field* to designate the field to be totaled.

- ❗ Do not use SUM to total the value of a field that contains non-numeric data. This will generate an error message for each record processed.

options

Options must be enclosed within parentheses. More than one option can be used within a single set of parentheses. Two options are available:

Option	Meaning
(P)	Use the P option to send the total number of records counted to the printer.
(S)	Use the S option to suppress message output.

Correct Use of SUM Command

SUM CUSTOMERS AMOUNT.DUE WITH DAYS.OUTSTANDING OVER "45"
The SUM command first selects those records in the file CUSTOMERS that have a value greater than 45 in the field DAY.OUTSTANDING, and then totals the values in the field AMOUNT.DUE.

SELECT CUSTOMERS WITH DAYS.OUTSTANDING > 45
SUM CUSTOMERS AMOUNT.DUE
Produces the same result as the previous example, but uses the SELECT command to first create an active select list of those records in the file.

SUSPEND

The SUSPEND command temporarily halts an Advanced Revelation operation to call any operating system command or to enter the operating system environment.

SUSPEND {EXIT} {oscommand}

Using SUSPEND

Use SUSPEND to execute external programs (such as spreadsheets, word processors, utilities, etc.) from within Advanced Revelation, while releasing almost all RAM occupied by Advanced Revelation.

SUSPEND writes the current status of Advanced Revelation to a file. The file to which SUSPEND writes Advanced Revelation is determined by a user's environments or can be set with the ROLLOUT command. If no ROLLOUT filename is specified, SUSPEND will act like the TCL command PC.

☞ In OS/2 versions, SUSPEND and PC are identical in operation.

To use SUSPEND, the operating system command processor must be accessible. Under MS-DOS and OS/2, this may be accomplished indirectly by using the SET COMSPEC command.

You can change the default drive while in the operating system environment, but Advanced Revelation resets the default to its initial value when you exit. To return to Advanced Revelation from the operating system, type EXIT at the operating system prompt.

Warning! Do not execute any memory-resident utilities while Advanced Revelation is suspended.

Advanced Revelation is delivered with @ROLLOUT.FILE empty. Before using the command SUSPEND or the R/BASIC statement SUSPEND, you must set up an @ROLLOUT.FILE using ROLLOUT.

EXIT

Preceding *oscommand* with the EXIT option returns control directly to Advanced Revelation without prompting the user to press [Enter] first.

oscommand

Use *oscommand* to perform any specific operating system commands. For example:

```
SUSPEND CALC
```

This calls the operating system batch file CALC, which runs a spreadsheet program. The program writes Advanced Revelation to a newly created operating system file before the spreadsheet begins. When the spreadsheet operations are complete, you are prompted to press [Enter] to return to Advanced Revelation.

If you do not specify an *oscommand*, SUSPEND leaves you at the operating system command line.

See your operating system manual for further command information.

Correct Uses of SUSPEND Command

```
SUSPEND
```

```
The IBM Personal Computer DOS
Version 3.1 (C) Copyright IBM Corp 1981, 1982, 1983, 1984
```

```
C:>chkdsk B:
```

```
362496 bytes total disk space
 37888 bytes in 2 hidden files
134144 bytes in 50 user files
190464 bytes available on disk
```

```
654336 bytes total memory
5955424 bytes free
```

```
C:>exit
```

The program leaves Advanced Revelation, enters MS-DOS and displays the C:> prompt, signaling its readiness for an MS-DOS command. MS-DOS processes the CHKDSK command on disk drive B:, completes the operation, and again displays the C:> prompt. Typing the MS-DOS EXIT command returns the Advanced Revelation program to its initial default drive and program location.

TERMINAL

The **TERMINAL** command turns the PC into a terminal that can be used with a mini or mainframe computer.

- ☞ In OS/2 versions, terminal emulation is not currently supported. Using **TERMINAL** will have no effect.

TERM is a synonym for **TERMINAL**.

TERMINAL {settings}

Using TERMINAL

Use **TERMINAL** to make the PC emulate an ADDS Regent 100 terminal. Executing the **TERMINAL** command immediately puts you into terminal emulation mode. Use [F9] to return to Advanced Revelation.

settings

TERMINAL can change the default communications settings (displayed at the Environment window). To do this, include one or more switch-setting parameters directly in the command line. Each switch specified must be preceded by a dash (-). The valid switches and settings are:

Table I 3-1
Terminal Emulation Settings

Switch	Setting	Description
-BAUD	<i>n</i>	Sets the baud rate to <i>n</i> (default = 9600)
-BITS	<i>n</i>	Sets the number of data bits to <i>n</i> (default = 7)
-STOP	<i>n</i>	Sets the number of stop bits to <i>n</i> (default = 1)
-PARITY	E/O/S/M/N	Sets parity to E(ven), O(dd), S(pace), M(ark), or N(one) (default = O)
-XOFF	yes/no	Enables/disables xon/xoff protocol (default = NO)
-DEVICE	<i>name</i>	Uses the installed device driver identified by the internal name (default = COMM\$, the COMMDRV.IBM internal name)

(continued)

Table I 3-1

Terminal Emulation Settings (Continued)

Switch	Setting	Description
-MARK		Translates ASCII characters with the hex value of 7B-7F to ASCII characters FB-FF. This permits the transmission of Advanced Revelation system delimiters over 7-bit communications lines.
-BREAK	<i>n</i>	Sets the break delay value to the value <i>n</i> , which must be between 1 and 255 (default = 1)
-STRIP/-NOSTRIP		Enables/disables the stripping of leading nulls from host transmissions (default = NOSTRIP)
-LOOK		Reports the current switch settings without changing them and before entering terminal emulation mode

Setup and Operation

In order to use the **TERMINAL** command, your system must have an Advanced Revelation communications driver installed on your boot drive. The communications driver is in the file **COMMDRV.IBM** and is located in the directory at which you installed Advanced Revelation.

You will need to edit the **CONFIG.SYS** file of your system and add this line:

```
DEVICE = COMMDRV.IBM n,m
```

where *n* is the number of bytes to be allocated to the communications buffer (default = 1024), and *m* is an integer value representing the serial port (COM1, COM2, etc.) to be used by the driver (default = 1).

For example, to load the device driver for communications port 2 with a 4K buffer, add this line to **CONFIG.SYS**:

```
DEVICE = COMMDRV.IBM 4096,2
```

When **CONFIG.SYS** has been changed, reboot your workstation.

Downloading data from the host

Data can be downloaded from the host directly into an Advanced Revelation file by including a special string of characters in front of each downloaded record. The format of the data to be downloaded is shown in this example being assigned to the variable `STRING`:

```
STRING=CHAR(27):"RF":filename:",":key:",":length:",":data
```

where:

- `CHAR(27)` is the ASCII character for ESC
- `"RF"` is the literal string RF
- *Filename* is the name of the Advanced Revelation file that will receive the data
- *Key* is the key under which the downloaded record will be written when entering terminal emulation
- *Length* is the length of the data record
- *Data* is the data constituting the record

A program should be written for the host to precede each record being downloaded with the above information.

Correct Use of TERMINAL Command

`TERMINAL`

The workstation is now ready to be used as a terminal.

`TERM -LOOK`

Advanced Revelation reports the current communication settings as it enters terminal emulation mode.

`TERMINAL -BAUD 1200 -BITS 8 -PARITY E`

Sets the baud rate, data bits and parity of the communications port.

TIME

`TIME` returns the current time and date.

TIME

Using TIME

Use `TIME` to display a window containing the current time and date.

Correct Use of TIME Command

TIME

A window is displayed showing the time and date. Press any key to leave the window.

TRANSACTION

The TRANSACTION command controls transaction processing for the current workstation. TRANSACTION may be used with:

- SQL tables
- Any file that has had transaction controls added with the CONTROL_ON command

For more information about transaction processing, see the “Managing Files and Records” chapter of *Filing Systems*.

TRANSACTION	ON COMMIT ROLLBACK RESTART STATUS	{option}
-------------	---	----------

ON

The ON keyword activates transaction processing.

COMMIT

The COMMIT keyword commits the current transaction.

ROLLBACK

The ROLLBACK keyword aborts (rolls back) the current transaction.

RESTART

The RESTART keyword restarts the most recent transaction aborted during a commit operation.

STATUS

The STATUS keyword displays status information for all transactions at the workstation. The following transaction information is displayed:

- File information for active transactions
- Aborted transactions
- Time and date

- An activity index indicating the relative amount of work done by a transaction
- The auto-commit and auto-transact settings

option

The (S) option (enclosed in parentheses) suppresses the display of error message or status line information.

Correct Use of TRANSACTION Command

TRANSACTION ON

Transaction processing is enabled for the workstation.

TRANSACTION RESTART (S)

Restarts the most recent aborted transaction. Messages and status line information are not displayed.

Incorrect Use of TRANSACTION Command

TRANSACTION

A parameter is required.

TUTORIAL

The TUTORIAL command activates the Advanced Revelation tutorial.

TUTORIAL

Correct Use of TUTORIAL Command

TUTORIAL

The Advanced Revelation tutorial menu displays.

UNMAKEVOC

The UNMAKEVOC command removes the VOC file reference pointer to an R/BASIC program.

UNMAKEVOC program {option}

Using UNMAKEVOC

UNMAKEVOC removes the entry in the VOC file established by MAKEVOC.

UNMAKEVOC will only delete VOC entries that contain the word RBASIC in the first field.

program

The name of a cataloged R/BASIC program

option

Use the option (S) enclosed in parentheses to suppress message output.

Correct Use of UNMAKEVOC Command

```
UNMAKEVOC UPDATE.CUST INVERT
```

This command removes the VOC file reference pointers for the R/BASIC programs UPDATE.CUST and INVERT.

WHO

WHO produces a report of the current user and account names, system environment, and network environment.

- For more information on users, accounts, and security, see the chapter “Users and Accounts: Planning Access and Security” in *Developing an Application*.

WHO

Using WHO

The WHO report consists of three pages of information. Page one displays the following:

Serial #	The serial number for the copy of Advanced Revelation currently booted.
Release Version	The subrelease version currently running.
User Name	The name under which the user has logged in.
Account Name	The account associated with the current user or the account into which the user has logged.
Restriction Level	The system security level established for the current user in the current account.
CPU	The CPU installed in the workstation.
Math Processor	Indicates whether a math coprocessing chip is installed.

Available Memory	The current amount of free space (string space) available in RAM.
Descriptors Used	The number of variables currently initialized for all resident Advanced Revelation programs.
Vendor/Type	The brand or type of network under which Advanced Revelation is running.
Maximum Users	The number of concurrent network sessions which this copy of Advanced Revelation will allow.
Station Id	The identifier by which the network is identifying the current station.

Page two of the WHO report displays the following information:

Optional Module Information	The modules may be any of the optional snap-on modules available (an example might be the Environmental Bonding modules). The module information consists of: Module name Release version Date installed	
Upgrade Information	The upgrade information describes any currently installed Advanced Revelation upgrades. From Level The previous version number. To Level The current upgrade version number. Date Installed	

Page three of the WHO report displays extended memory information.

Correct Use of WHO Command

WHO
Displays a WHO report to the screen.

WINDOW

The WINDOW command displays application and command windows from TCL.

```
WINDOW {@filename@} template {TABLE.ON} ⇐  
  {PREAPP=@code command@} {DATAFILE=filename} {keys}
```

Using WINDOW

When the window name is not specified, a specification window is displayed. Type the window name at the entry line of this window.

To read a window template from another file, type the file name enclosed by at-signs (@). If no file name is specified, the window template is read from the TEMPLATES file. A window template can be stored in a file other than TEMPLATES.

A command window template is stored in the COMMANDS file. The command CW is provided for displaying command windows. For example:

```
WINDOW @COMMANDS@ RECORDCOPY
```

and

```
CW RECORDCOPY
```

perform the same function.

See the CW command for more information.

template

The *template* must be the name of a window template record.

TABLE.ON

TABLE.ON can be used to display the window in table mode. Table mode provides a display similar to a spreadsheet. Table mode allows the display of more than one record at a time.

PREAPP=@code command@

PREAPP=@code command@ will execute a code and command before the window is displayed. You can use more than one PREAPP code and command in a statement.

DATAFILE=filename

DATAFILE=*filename* can be used to change the file which the window updates or reads.

keys

Keys can be used to create a browse list for the window. List the record keys, separated by spaces.

Correct use of WINDOW Command

```
WINDOW CUSTOMER TABLE.ON
```

The CUSTOMER window is displayed in table mode.



Chapter J 1

 Introduction to R/LIST

With Advanced Revelation’s R/LIST, you can retrieve data and produce reports using plain English. Combine simple R/LIST commands with a host of options to produce reports in the format that suits your needs.

Introduction to R/LIST	J 1 . 1
Using Filters	J 1 . 6





Introduction to R/LIST

R/LIST is Advanced Revelation's English-like retrieval and reporting language. It is a free-form language that enables users to produce reports using natural commands and syntax.

R/LIST commands include options to find a subset of records on file (filtering), arrange the records into a specified order, and choose certain fields to display. Reports can include control breaks, totals and subtotals, averages, headers and footers. R/LIST also offers various options to control line spacing, inclusion and suppression of titles, printer or screen output, and the generation of R/BASIC source code. Most importantly, you can include keywords with R/LIST commands to filter the data used in the R/LIST operation.

R/LIST and Dictionaries

R/LIST commands are parsed word by word from the command forward. Advanced Revelation will look first into the dictionary of the file being queried.

- ☛ The USING keyword can redirect the dictionary lookup to the dictionary of another file.

Normally, the fields used to select, sort and display data will be found in this local dictionary.

If a word is not found in the local dictionary, Advanced Revelation will look into the dictionary of the VOC file. As noted, most R/LIST commands are defined in this dictionary.

The fact that Advanced Revelation will always look in two dictionaries means that the dictionary of the VOC file can be used to store field global definitions. For example, the dictionary of the VOC file contains the definition of a field called F1. This field is defined simply as "field one". Any R/LIST command can reference the field F1, and R/LIST will extract the first field of records being displayed based on this global definition.

- ☛ R/LIST searches are always case-sensitive (uppercase and lowercase characters are distinguished). This is true whether you define your indexes as case-insensitive or not.

R/LIST Commands

The following are the R/LIST commands available at TCL:

- LIST** Produces a report directly to the screen or printer. Records are output in rows, and fields from the record are displayed in columns across the page.
- SORT** Like LIST, but also always sorts the report into record key order.

COUNT	Produces a report of the number of records that meet specified criteria.
SUM	Totals the contents of a field for a set of records meeting specified criteria.
SELECT and SSELECT	Creates and activates a filter (list of record keys).
Rlist	Activates a predefined LIST template.

Capturing Source Code

The LIST and SORT commands generate, compile, and execute an R/BASIC program each time they are executed. You can capture the R/BASIC source code for the report and use it in your own applications.

Capturing source code is useful when you have reporting requirements that cannot be met by using R/LIST's capabilities or by using symbolic dictionary items. You can use the R/LIST source code generated by LIST as the basis of the report, and add modifications as required.

Once you have made modifications to the source code, you can compile and run the program. Note that the program you execute will not run faster than R/LIST itself. The report will not, however, execute the pause that R/LIST requires while it generates the source code.

- ✎ LIST and SORT will generate the same source code; SORT will not generate code used to sort data in the file.

LIST and SORT Options To capture R/LIST source code generated by SORT or LIST, use either of the two following options.

- (X) The (X) option will cause LIST or SORT to capture source code and then execute the report. It brings up a window with prompts for the file name and the record name. Enter the name of the file and the record key under which you wish to store the captured source code.
- (SX) The (SX) option will cause LIST or SORT to capture source code without executing a report.

Unlike the (X) option, (SX) does not prompt you for a location to store source code. It will always write the source code to a native file named RLIST. If you generate more than one report, the record will be overwritten with each new one. Use RECORDCOPY to copy the source code to another file before generating more source code.

For example, to copy the source code to the BP file under the name CUST.RPT, use this command:

```
RECORDCOPY DOS RLIST TO: (BP CUST.RPT
```

Creating Synonyms

To make R/LIST easy to use, you can define synonyms for R/LIST commands, keywords, and throwaway words. For instance, a user or developer may discover that it is more natural to use the word “fetch” rather than the word “list,” or that the word “when” is preferable to the word “with”.

Use the RECORDCOPY command to create synonyms. RECORDCOPY duplicates the command or keyword under a new name. Once the original command or keyword has been copied, it is treated exactly like the original.

The file to use with RECORDCOPY depends on whether you are creating a synonym for a command or a keyword. Commands (LIST, SORT, SUM, COUNT) are stored in the VOC file, while keywords and throwaway words are stored in the dictionary of the VOC file (DICT.VOC).

The syntax for using RECORDCOPY to make a synonym is:

```
RECORDCOPY VOC command TO: new.command
```

or

```
RECORDCOPY DICT VOC keyword TO: new.keyword
```

For example, the following command will make “VIEW” a synonym for the command LIST.

```
RECORDCOPY VOC LIST TO: VIEW
```

This command will make a synonym “PRINTER” for the keyword LPTR:

```
RECORDCOPY DICT VOC LPTR TO: PRINTER
```

Inline Formulas

You can include formulas directly in the R/LIST command line. In addition to displaying data from the fields listed in your R/LIST command, you can create calculations or other formulas to be processed in your report.

The formulas you create in the R/LIST command function just like symbolic fields created in the dictionary of the file. Inline formulas can execute mathematical computations, reformat data already on file, join or translate data from other files, or perform any other function available to symbolic formulas.

For more information on creating formulas, see the “Building Formulas” chapter in *Developing an Application*.

Creating an Inline Formula

Inline formulas can be inserted anywhere into an R/LIST command where you might otherwise have a field name. To indicate that you are including an inline formula, surround your formula with {} (braces). Inline formulas can be used in place of field names for display, for selecting or for sorting. Examples of the latter two appear in “Using Filters” later in this chapter.

Within the braces, simply write out the calculation or other function you wish to perform. The same mathematical operators and R/BASIC functions apply that you might use in a formula for a symbolic field. For example, note these two R/LIST commands:

```
LIST CUSTOMER ADDRESS { "Attn: ":{CONTACT} }
```

```
LIST PRODUCT { ({QTY.ON.HAND} + {  
{ON.ORDER}) - {BACK.ORDERED} }
```

The first example includes an inline formula that inserts the word “Attn:” in front of the contact name. This example illustrates that you can reference other fields by surrounding the name of the field with another set of braces {}. This follows the conventions used for writing formulas in the dictionary.

The second example adds together the values of the fields QTY.ON.HAND and ON.ORDER, and subtracts from that sum the value of the field BACK.ORDERED. Parentheses are used in this formula to indicate mathematical priority, just as in a dictionary formula.

In both cases, the expression or calculation used to define the formula is simply typed in. It is not necessary for simple inline formulas to assign the result of a formula to any particular variable. This is different from dictionary formulas, in which it is necessary to assign the value of a formula to the system variable @ANS.

Multiline Inline Formulas

If your inline formula is complex enough to require more than one line of logic, you can create a multiline formula. To do so, separate each line of your inline formula with ; (semicolon).

Unlike simple inline formulas, complex inline formulas require that you assign the value of the formula to @ANS. At the point in your formula where you have calculated the answer, assign the value to the system variable @ANS.

For example, the following multiline inline formula explicitly sets the value of @ANS:

```
LIST INVOICE { PRICE = {PRICE} ; ⇨
IF {DISCOUNT} THEN PRICE = PRICE * ⇨
{DISCOUNT} ;@ANS = PRICE }
```

In this example, the inline formula calculates a price based on a discount. If a discount is available, it is applied to the price. Whether discounted or not, the price is assigned to the system variable @ANS at the end of the formula. Each line of the formula is delimited with ; (semicolon).

Output Specifications for Inline Formulas

Because inline formulas are created on-the-fly, you do not have a chance in advance to specify format specifications such as justification, length, conversions, or column headings. The inline formula processor therefore assumes certain values for these fields. You may either accept the defaults, or override the format specifications.

Defaults Unless you specify otherwise, an inline formula will use the following values for output specifications:

Specification	Default Value
JUSTLEN	L and 10
OCONV	(none)
COLHEAD	Temp Formula <i>n</i> (<i>n</i> is a system-assigned number)

Overrides To override these default values, use the Format Override keywords available in R/LIST. For example, the following R/LIST command assigns a column heading and an output conversion to an inline formula:

```
LIST INVOICE { {INV.DT}+30 } COLHEAD ⇨
"Due Date" OCONV "D2/"
```

In the example, the inline formula calculates a value that is 30 days from the invoice date value. The column heading "Due Date" is displayed above the column, and the data is output with a date conversion.

Creating Fields from Inline Formulas

You can create a permanent field in the dictionary out of an inline formula. This enables you to capture an inline formula and write it to the dictionary as a symbolic field.

To create a field out of an inline formula, use this syntax:

```
LIST filename new.field.name:={formula}
```

The new field name precedes the inline formula, and is assigned the formula with the `:=` operator. The field name in the R/LIST command can be any name that is a valid field name in the dictionary. If the field name is already defined in the dictionary, you will be prompted to overwrite the old version of the field.

For example, the following inline formula assigns an inline formula to the new field DUE.DT:

```
LIST INVOICE BY DUE.DT:={ {INV.DATE}+30 } ⇨  
DUE.DT
```

The new field is created in the sort clause, and is reused as a display field.

Using Filters

Filtering is the process of creating a subset of records out of all the data in a file. When a subset of records has been created, a “filter” is said to be active. Under most circumstances, once a filter is active, the next command to be executed will process only the records represented by the filter.

The subset of records can represent a selection of records from the file. It can also represent an ordered (sorted) list of records, or it can represent both.

In actuality, a filter is simply a list of record keys. The keys represent the records in a file constituting the subset or sorted set of data to be processed.

Filters are “active” if the list of keys is in a queue in memory. If a filter is active, most record-oriented TCL commands (EDIT, RECORDCOPY, DELETEREC, or BLIST for example), and all record-oriented R/LIST commands, will process only the records whose record keys are in the list. Filtering is therefore a two-step process: creating and activating the list of record keys, then passing them to a subsequent command for actual processing.

If a filter is active, the status line indicates this by displaying the word “filter”. As the list of record keys is processed by the command, it is used up. When the command has processed all records on the list, the list is no longer active.

Using Filters at TCL

There are several R/LIST commands that can create, save or retrieve lists of record keys. In addition, the commands LIST, SORT, COUNT and SUM can create filters as they execute.

The R/LIST commands SELECT and SSELECT create and activate lists of record keys. These commands can include selection criteria to specify which records are to be put into the list of record keys for the filter, and sort criteria to create an ordered list of keys.

The R/LIST commands GETLIST and MAKELIST can turn data on file into an active filter. These are used when lists of record keys have been stored on file by another process. The R/LIST command SAVELIST can store an active list of record keys to file for later retrieval.

Using Filters with R/LIST Keywords

The R/LIST commands LIST, SORT, COUNT and SUM can create their own filters. You can specify this by including selection and sort criteria in the same manner as you would with a SELECT statement.

Simply precede the R/LIST keyword(s) in your command with selection or sort criteria. The R/LIST process will then use the filtered data.

See the appropriate R/LIST command in the "TCL Command Reference" chapter of *TCL* for proper syntax.

Creating Active Filters

SELECT and SSELECT create and activate filters for use by subsequent commands.

The format for SELECT and SSELECT is:

```
SELECT filename {selection.criteria} {sort.criteria}
```

An example of a SELECT statement might be:

```
SELECT INVOICES WITH INV.DATE "1-1-87" BY INV.DATE
```

The *selection.criteria* determine which records in the file are to be included in the filter. The *sort.criteria* determine the order of the records to be processed.

- ☛ For more information about SELECT and SSELECT, see "TCL Command Reference" in *TCL*.

The Selection Criteria

The *selection.criteria* can take two forms: a list of explicit record keys, or a WITH clause.

If the command includes explicit record keys, the keys must be enclosed in quotes. If explicit keys are included, SELECT or SSELECT will simply activate that list directly; the command will not need to access the data file or any indexes. For example, the command:

```
SELECT CUSTOMERS "C100" "A104" "P477"
```

will create a filter out of the three customers whose keys are listed in quotes in the command.

Alternately, the *selection.criteria* can consist of a WITH clause. The WITH clause establishes a specification or definition for records that are to be included in the filter. For example, the command

```
SELECT INVOICES WITH INV.DATE > "1-1-87"
```

indicates that the filter is to consist of all records in the INVOICES file in which the invoice date is greater than January 1, 1987.

If a WITH clause is used as the *selection.criteria*, the command will either need to access an index (if the field being searched has been indexed), or review each record in the file individually to see if it matches the *selection.criteria*.

For more information about the WITH clause, see "Filter Keywords" in the "R/LIST Keyword Reference" chapter.

The Sort Criteria

The *sort.criteria* cause the filter to be ordered according to the contents of one or more fields. Note that including *sort.criteria* does not cause a physical sort of data in the file. Instead, the list of keys represents how the data would appear if it had been sorted according to the specifications.

For more information about the use of the *sort.criteria*, see the Filter keyword BY in the "R/LIST Keyword Reference" chapter.

Saving and Recalling Record Key Lists

Frequently it is useful to be able to use the same filter for more than one command. Instead of having to regenerate the filter each time, you can save an active filter away to file for later recall.

The command SAVELIST stores an active filter on file. When the record key list is stored, it is given a name by which it can be recalled later.

The command GETLIST retrieves and activates a list previously stored with SAVELIST.

The MAKELIST command can make an active filter out of any record or any multi-valued field in any record. MAKELIST is used primarily to create active lists out of fields being maintained by a Relational Index.

For more information about each of these commands, see the "TCL Command Reference" in *TCL*. For information about Relational Indexing, see the *Indexing* section.

Percentage Done and Status Bar

The status line keeps you advised about the progress of your filter operation. As Advanced Revelation works on filtering your data, it prints two messages at the top of the status line.

The first of these is a Percentage Done number. This number is updated as the filter process executes, providing you with a clear indication of how far along the operation is.

In addition to a Percentage Done number, you will also see a graphic Status Bar. The Status Bar is a bar graph representation of the number in Percentage Done. As the filter operation proceeds, the Status Bar grows to the right. When it reaches the right margin of the status line, the filter operation is complete.

Conditions for Displaying Percentage Done

As long as the status line is active, the filter process will generally create a Percentage Done counter and the Status Bar. Certain conditions will prevent the Percentage Done counter and Status Bar from displaying. Generally speaking, these two messages are not displayed if Advanced Revelation determines that the filter operation will not last very long. Among the conditions that prevent the messages from displaying are these:

- A report is displaying in the View window.
- The file contains less than 100 records.
- The filter is based on one or more indexed fields.
- The filter is not based on a file created under Version 1.1 or later of Advanced Revelation, and this is a new filter (not working against an active select list).

If the status line has been turned off completely, the messages will not appear.

Inline Formulas for Filters

You can use inline formulas to calculate values used in a SELECT statement. This enables you to sort or select on values that are calculated or looked up on-the-fly, without having to create a special symbolic field. Inline calculations can be substituted anywhere in a SELECT command where you would otherwise have a field name.

For example, you can use an inline formula to create a value on which you base a selection:

```
SELECT INVOICE WITH { {PAID.DATE}- {DUE.DATE} } > 90
```

In this example, records in the INVOICE file will be selected if the difference between the invoice due date and the actual paid date is greater than 90 days.

You can also use inline formulas as the basis for sorting. To do so, substitute an inline formula for a field name after the word BY. The following example illustrates this:

```
LIST INVOICE BY {XLATE("CUSTOMER",{CUST.NO},⇨
"CITY", "X") }
```

In this example, the data in the INVOICE file is sorted on the value of the customer's city. The inline formula looks up the customer's city based on the value of the customer number. The resulting value is then used in the sort.

Conversions for Inline Formulas

In some instances, you may find that the data calculated by an inline formula is returned in internal format. This will typically happen if you are calculating or looking up a value that concerns money, date or time.

You can use the OCONV keyword with an inline formula in order to cause the calculated value to be converted before processing. In other words, if you create an inline formula, you can use OCONV immediately following the formula in order to format or convert the data before that data is used in a selection clause.

☞ Format override keywords are discussed in the “R/LIST Reference” chapter of *R/LIST*.

The following SELECT command illustrates the use of the format override keyword OCONV with an inline formula:

```
SELECT CUSTOMER WITH {SUM({INVOICE.AMT})} ⇨
OCONV "MD2,$" > 1000
```

In this example, records in the CUSTOMER file will be selected only if the sum of the values in the invoice amount field exceeds \$1,000.00. The inline formula actually returns data in internal format (for example, \$1,000.00 is returned as 100000). However, the format override for the conversion is applied before the value of the formula is compared to the data value “1000”.

Justification for Inline Formulas

Inline formulas always default to left justification. This justification is significant when basing a sort on an inline formula.

If you create an inline formula that results in numeric data, and if you are using that inline formula as the basis for a sort, you should use the format override keyword JUSTLEN to change the justification of the formula to R. If the keyword JUSTLEN follows an inline formula in a BY clause, the justification override will be used in sorting the data.

For example, the following SELECT command uses an inline formula as the basis for a sort, and uses a format override keyword to change the justification of the sort:

```
SELECT CUSTOMER BY { SUM({INVOICE.AMT}) } ⇨  
JUSTLEN "R"
```

In this example, the inline formula calculates a value that is the sum of the invoice amount field. The filter is sorted by the value of this inline formula. The format keyword JUSTLEN "R" overrides the default left justification of the inline formula, so the value of the inline formula will be sorted with a right justification.

Resolved and Latent Lists

You have an option in creating select lists that can make the process of selecting and processing data much more efficient. A second type of select list, a 'latent' list, bypasses the requirement to assemble all record keys in advance. There are special keywords for R/LIST commands that give you partial control over the creation of latent lists.

Resolved Lists

The process of using a filter (select list) consists of two steps: selecting the data and putting it into a select list, and then passing the list to a process that uses it. The first function is accomplished with a SELECT command (from TCL, from the Filter key, from Query mode, from a stored filter, or from within an R/BASIC program). The process using the list can be any record-oriented command (LIST, or EDIT for example) or an R/BASIC program.

Typically when a select list is created using a SELECT command, the resulting list of record keys is resolved. This means that the file or index has been reviewed, and the list of keys assembled. The process that follows the SELECT command can then simply read the keys from the list and process them.

A resolved list is static. Once the list is created, subsequent processes do not update the list. For example, once a resolved list has been assembled, no further keys are added to the list even if the file is updated. Conversely, no keys are removed from the list, even if records represented by those keys are deleted from the file.

In order to create a resolved list, the selection process must either review the entire file or the appropriate index. If no indexes are involved in the selection, the filter process must read each record in the file and review it to see if it meets the selection criteria.

The result is that records involved in the filter (those being selected) might need to be read twice. The first time is when the filter logic reviews them for inclusion in the select list. The second is when the record-oriented operation that requires the select list (such as LIST, EDIT, or an R/BASIC program) actually processes the record.

Latent Lists

A way to avoid reading records twice is to create a latent select list. A latent list is one in which the keys meeting the selection criteria are not actually assembled in advance. Instead, the criteria are set aside. When the subsequent record-oriented operation is executed that operation reads through the file or index and applies the selection criteria to each record. If a record meets the selection criteria, it is processed. If it does not meet the criteria, it is skipped.

The net effect is that the creation of the select list and the processing of the records in the list is compressed into a single operation. Rather than approaching the steps of selecting and processing records as two separate steps, this compression accomplishes both at once.

Using a latent list thus saves the time of reading records twice, making the entire procedure faster. It is important to realize, however, where the speed is gained.

If you establish a latent list, almost no time is required to “select” the file. This is because the system is not actually creating a list: it is just creating and setting aside the selection and sort criteria.

On the other hand, the operation that follows will probably take longer than it would in working with a resolved list. The second operation must spend extra time in applying the filter criteria to data as records are processed.

Speed is gained in the first operation (selecting the records), but lost in the second (processing the records). In absolute terms, however, the complete process of selecting and then processing data is less time-consuming when you use a latent list.

When to Use Resolved Lists

Because latent select lists are typically more efficient than resolved lists, you will probably want to create latent lists most of the time. There are just a few circumstances in which creating a resolved list is the better strategy.

The first is if you intend to update the file while processing the list. In this case, a resolved list might be a safer way to process, since the select list is static.

If you are using a latent list in a process that updates the file, you may be creating new records. The keys for these new records might be fed into the latent list. As a result, the latent list will be dynamic, and it will not be predictable which keys will appear in the list. A resolved list avoids this problem, since the list of keys to be processed is assembled in advance.

A dynamic latent list might not always be undesirable. However, since you cannot predict which records will be processed if you are writing to the file, it is advisable to avoid latent lists for files being updated.

A second reason to use resolved lists is to optimize the processing of the records (as distinct from selecting them). If you are using a latent list, the process using the list must not only process the records, but must determine whether the records belong in the select list. The process actually using the select list (the R/LIST command or R/BASIC program for example) therefore executes slower when using a latent list than when using a resolved list. Viewed as a whole, the cycle of selecting and processing will be faster, but the latter process, the one using the select list, will run a bit slower.

If you are not concerned about the time spent selecting the file, but do care how fast the actual processing operation executes, use a resolved list. For example, you might select records from a file overnight, when no other processing is being done on a file. You can then run a LIST or other process during the day. Because the LIST or other process is not spending time in resolving the latent list, it will execute at optimal speed.

Creating Latent Lists

There are two keywords that you may use with R/LIST commands to request that a select list created is to be a latent list. These two words are the synonyms NORESOLVE and LATENT.

LATENT and NORESOLVE

In specifying LATENT or NORESOLVE, you will be overriding the default method, which is to create a resolved select list.

The new keywords are only advisory. The filter process cannot create a latent list for all select lists. It will therefore only honor your request for a latent list if the request does not contradict the rules for resolved lists.

For example, the following SELECT command includes the advisory keyword NORESOLVE:

```
SELECT CUSTOMER WITH COMPANY "[ACME]" ⇔
AND WITH ZIP > "90000" NORESOLVE
```

Unless forced to use a resolved list, this command will result in a latent select list. The next command executed after this SELECT command will resolve the latent list as it processes.

Rules for Resolved Lists

Certain criteria will always force the R/LIST command to resolve the select list. This will happen even if you include the advisory keywords for creating a latent select list.

The following circumstances will always create a resolved select list:

- You have specified a count limit on the number of records to process. For example, the following command incorporates a count limit:

```
SELECT 10 CUSTOMER WITH ST = 'WA'
```

- All fields involved in the selection criteria are indexed. If this is the case, it is just as efficient for the command to resolve the list as to create a latent list.
- The first or only sort field is not indexed.
- The first or only sort field is multivalued.

Status Line Indicator

If the select list you create is resolved, the indicator “Filter” in the status line will be in uppercase letters (FILTER). If you have created a latent select list, the filter indicator will be in lowercase letters (filter). This will signal whether a list you have created is resolved or latent.

Chapter J 2

R/LIST Keyword Reference

R/LIST enables you to retrieve data and create reports using a handful of plain-English commands and a simple, natural syntax.

This chapter explains how to use R/LIST's commands and formatting options to easily access your data and display or print it in a variety of configurations.

R/LIST Command Format	J 2 . 1
R/LIST Format Modifier Keywords	J 2 . 3
R/LIST Format Override Keywords	J 2 . 5
R/LIST Throwaway Words	J 2 . 6
R/LIST Keywords	J 2 . 6
AVERAGE	J 2 . 7
BREAK-ON	J 2 . 7
BY	J 2 . 9
COLHEAD	J 2 . 10
FOOTING	J 2 . 11
GRAND-TOTAL	J 2 . 12
HEADING	J 2 . 13
JUSTLEN	J 2 . 15
LIMIT	J 2 . 16
OCONV	J 2 . 18
ONLY	J 2 . 18
TOTAL	J 2 . 19
USING	J 2 . 20
WITH	J 2 . 21



R/LIST Command Format

The primary R/LIST commands share a general format, one that allows a very loose ordering of keyword clauses for a more natural usage.

The control keywords **ONLY** and **USING** are exceptional in this regard. **ONLY** must appear just before or after the file specification, while the **USING** clause may appear only immediately after the file specification. This is illustrated in the format below.

The specifications for the various keywords and information on how to use them may be found later in this chapter.

```
command {DICT} filename {USING alt.filename} ⇨
  {selection.criteria} {sort.criteria} ⇨
  { {control.keyword} display.field {format.override} ::} ⇨
  {format.keyword} {format.modifier} {options}
```

command

Command may be any of the following R/LIST commands:

LIST
SORT
COUNT
SUM
SELECT or SSELECT

☞ For more information about these commands, see the chapter “TCL Command Reference” in *TCL*.

DICT

If **DICT** is specified, the report will be generated against the dictionary portion of a file.

filename

The *filename* can be any attached Advanced Revelation file. *Filename* is the only word other than the command itself that is required.

USING

A **USING** clause specifies that the operation is to use the dictionary of the alternative file *alt.filename*, rather than the *filename* dictionary.

selection.criteria

If *selection.criteria* are specified, the *command* will process select a subset of records from *filename* for processing. The *selection.criteria* are either a list of record keys or a WITH clause (see the keyword WITH in this chapter).

sort.criteria

If *sort.criteria* are specified, the records are displayed in a particular order (see the keyword BY in this chapter).

control.keyword

The *control.keywords* affect the flow of control for the *command*. If specified, these may be any of the following:

AVERAGE	Produce a running average for a numeric value.
BREAK-ON	Produce a “break” in the report when the value of a field changes.
LIMIT	Display only particular values out of a multivalued field.
ONLY	List only the record keys in a report.
TOTAL	Produce subtotals and a grand total for a numeric field.
USING	Use an alternative dictionary.

Each of these keywords is documented later in this chapter.

display.field

If you are using the R/LIST commands LIST or SORT, *display.field* is the name of a field whose contents will appear in the report. You can display any number of fields in your report. Fields are displayed in the report according the order in which their names appear in the LIST or SORT command.

- ☛ The key field (@ID) of a file is always displayed in a report unless you use the Format Modifier Keyword ID-SUPP.

If the sum of the length of all the fields displayed in the report is wider than the screen or printer device, the line will wrap around the next line. If you are displaying your report on the screen, you can use the R/LIST Viewer window to create a window up to 255 characters wide for your report. See the chapter “Creating Reports with Advanced Revelation” in *Creating Reports* for details.

format.override

If specified, these keywords will override the display specifications for a field (as defined in the dictionary). The *format.override* keywords are:

JUSTLEN	Override the justification and display length of a field.
OCONV	Override the conversion (output format) of a field.
COLHEAD	Specify a new column heading for a field.

Each of these keywords is documented later in this chapter.

format.keyword

If specified, a *format.keyword* adds formatting features to a report or form. The *format.keywords* are:

FOOTING	Display text at the bottom of each page of the report.
HEADING	Display text at the top of each page of the report.
GRAND-TOTAL	Specify text that accompanies a grand total.

format.modifier

Use a *format.modifier* to alter the default format or output specifications. Format modifier keywords are discussed in detail under the topic "Format Modifier Keywords" following.

options

The additional display or output *options* that you may include vary according to the *command* specified. For example, the option "(P)" sends output to a printer. See the documentation for the particular R/LIST command in the "TCL Command Reference" chapter of *TCL*.

R/LIST Format Modifier Keywords

Certain keywords in R/LIST change the format or output specifications of the report. These format modifiers control such attributes of the report as spacing, column headings, pausing between pages, and printing to the printer.

Using Format Modifiers

The various format modifier keywords can be inserted anywhere into the R/LIST command (as long as they are not in the middle of a clause). If more than one format modifier is included in the command, they do not need to be specified together.

The format modifier keywords are:

COL-HDR-SUPP	ID-SUPP
DBL-SPC	LPTR
DET-SUPP	NOPAGE
HDR-SUPP or SUPP	

COL-HDR-SUPP

COL-HDR-SUPP suppresses the column headings (field titles) that are automatically printed at the top of each page for each field being displayed. It also suppresses the default heading and “records processed” message (see HDR-SUPP, below).

DBL-SPC

The modifier DBL-SPC causes a blank line to be printed between each record on the report. Double spacing a report can make it easier to read, especially if there are many multivalued or text fields being displayed.

DET-SUPP

Used with TOTAL and AVERAGE, DET-SUPP causes a value to be printed at the bottom of the report, and a subtotal or average to be printed at each control break. DET-SUPP will cause the report to suppress printing the detail about each record used to calculate a total or average, and print only the total or average and subtotal and subaverage.

HDR-SUPP or SUPP

If no heading has been specified, R/LIST reports will include a default heading of the date, time, and page number, and will include the message “xxx record(s) processed” at the bottom of the report. HDR-SUPP and SUPP suppress the default heading and record count.

ID-SUPP

LIST and SORT will always print the record key field in the leftmost column of the report. ID-SUPP causes this field not to be printed.

LPTR

The keyword LPTR causes a report to be printed on the current printer device. If PDISK has been executed before the report is run, the output will be routed to an operating system file.

NOPAGE

The keyword NOPAGE allows a report displayed on the screen to continue from start to finish without interruption. It suppresses the default break at the end of each screen that ordinarily would prompt the user to “press a key to continue”. NOPAGE has no effect on reports sent to the printer.

Examples of Format Modifier Keywords

LIST CUSTOMERS **ID-SUPP** COMPANY ADDRESS CITY **LPTR**

This command will print to the printer only the company, address and city fields (no record key field) of the CUSTOMERS file.

LIST PHONE.LIST BY NAME **COL-HDR-SUPP** NAME PHONE **DBL-SPC**

This report will print out names and phone numbers double-spaced, but will include no headings, column headings or "xxx records processed" message.

SORT INVOICES BY INVOICE.DATE BREAK-ON INVOICE.DATE TOTAL ⇨
INVOICE.AMOUNT **DET-SUPP**

This report will sort the invoices into date order, then display the invoices with a break at each new invoice date. Only the subtotals and total will be printed; no individual invoice amounts will appear.

R/LIST Format Override Keywords

Three keywords in R/LIST enable you to temporarily override the format specifications (justification, length, output conversion, and column heading) defined in the dictionary for a field. The capability to change the format of a field temporarily can save you the trouble of making changes in the dictionary before creating R/LIST reports.

General Format

The keyword and override specifications follow the name of the field in the R/LIST command to be displayed. Anytime text appears in the override specification, the specification must appear in quotes.

The keywords available to specify a format override are:

JUSTLEN	Override justification and/or length
OCONV	Override output conversion
COLHEAD	Override column heading

For example, the following R/LIST command includes a format override specification for the field COMPANY:

LIST CUSTOMER COMPANY **JUSTLEN "L40"**

In this example, the COMPANY field will be displayed with a left justification, 40 characters wide. The justification and length specified for the COMPANY field in the dictionary of the CUSTOMER file are ignored in this report.

Each Format Override Keyword is listed separately later in this chapter.

R/LIST Throwaway Words

To make R/LIST commands easier to construct and to read, you can include “throwaway” words in the command. These are words that do not affect execution of the command in any way, but that make the R/LIST command into a more natural, English-like sentence.

Following are the throwaway words that are available in R/LIST and examples of how they can be used:

Word	Example
A, AN, ANY	LIST ANY COMPANY
ARE	RECORDS ARE GREATER THAN...
DOES	LIST PAYROLL IF IT DOES CONTAIN...
FILE	SORT THE CUSTOMERS FILE...
FOR	...FOR THE COMPANY...
HAS, IS	SELECT PAYROLL IF IT HAS...
IT	SORT PAYROLL IF IT HAS...
ITEMS	SORT CUSTOMER ITEMS...
OF	LIST CUSTOMERS WITH STATE OF ...
SHOW, SHOWING	LIST CUSTOMERS SHOWING COMPANY PHONE ..
THAN	...IF IT IS GREATER THAN...
THE	LIST THE CUSTOMERS IF...

R/LIST Keywords

The rest of this chapter is an alphabetical listing of the keywords used in R/LIST commands. Each keyword entry includes a description, a diagram of the proper syntax for the keyword, and examples of the correct use of the word.

AVERAGE

AVERAGE is used to generate the average of a numeric field and display it at the end of the report. If a BREAK-ON has been specified, AVERAGE will also display subaverages at each break point.

AVERAGE *fieldname*

Using AVERAGE

Use AVERAGE to calculate an average for numeric fields such as dollar amounts. AVERAGE will calculate an average value and display the result at the end of the report. If BREAK-ON has been specified, AVERAGE will also output subaverages at each BREAK-ON. Note in the example that the BREAK-ON and AVERAGE fields can be different.

The final average appears at the end of the report, and is identified by three asterisks (***) in the leftmost column of the report. To substitute an option for the three asterisks, see GRAND-TOTAL.

Any number of fields can be averaged in the report. Note that AVERAGE can be used with the modifier DET-SUPP (detail suppress) to create reports with *only* average values.

Correct Use of AVERAGE Keyword

LIST CUSTOMERS BY ST BREAK-ON ST **AVERAGE INVOICE.BALANCE**

This command will list all CUSTOMERS by order of state, putting a break between each state. At each break, the average invoice balance for all customers in that state will be displayed. An average of all invoice balances will be displayed at the end of the report.

BREAK-ON

BREAK-ON causes R/LIST to put a physical break in the report as the value of a field changes.

BREAK-ON *fieldname {options}*

Using BREAK-ON

Use BREAK-ON with fields that have been sorted. For example:

LIST CUSTOMERS BY CUST.ST **BREAK-ON CUST.ST**

fieldname

Each time the value of the field represented by *fieldname* changes, the report will indicate (with spacing, special characters, etc.) that a new value is being printed.

options

Options can be specified that cause BREAK-ON to take specific action when a field changes value. The default action for BREAK-ON is to print a blank line, then three asterisks (***) in the column for the BREAK-ON field, then another blank line. The report then continues.

Multiple BREAK-ON fields are permissible. If more than one BREAK-ON is indicated, the report will break when the value of any one BREAK-ON field changes.

BREAK-ON is frequently used in conjunction with the TOTAL keyword. If TOTAL is specified, subtotals will be printed at each breakpoint in the report (see TOTAL for details).

The specific action that BREAK-ON will take when a field value changes is dependent on the options specified by the user. Options can be included with BREAK-ON by specifying them after the field.

Text options are enclosed within double quotes. Control options ('B' or 'UU') are single-quoted within double quotes. For example:

```
...BREAK-ON EMPLOYEE.NO "Year to date:'UU' "...
```

The following options are available:

- text** Use text to substitute for the three asterisks normally printed. The text will be truncated to fit into the column width of the BREAK-ON field.
- 'B'** BREAK: Denotes the field whose breaking value is to be inserted into the heading. Must be used in conjunction with the 'B' option in HEADING.
- 'D'** DATA: Suppresses redundant totals by suppressing the break data line entirely if the preceding data field contains only one line of data. Useful in condensing a report.
- 'L'** LINE: Suppresses the blank line that usually precedes the break data line. Use this when you wish to condense a report. This option does not override the 'U' option.
 - ☛ The BREAK-ON 'L' option suppresses a line, while the HEADING 'L' option causes a blank line to be inserted.
- 'P'** PAGE: Prints a new page each time the field breaks.
- 'U'** UNDERLINE: Prints a line over all subtotals.

- 'UU' UNDERLINE/OVERLINE: Prints a line over and under the subtotals.
- 'V' VALUE: Prints the value of the breaking field at the break line in place of the three asterisks.

BY

Use the filter keyword BY with R/LIST commands to generate sorted lists or reports.

```
BY fieldname
BY-DSND fieldname
```

Using BY

BY arranges the list or report according to the order of the contents of a field. BY does not sort the data in the file. Instead, it generates a list of record keys that represent how the data appears if sorted by a certain field.

To create a sorted in descending (reverse) order, use the keyword BY-DSND.

Multiple BY Clauses

Multiple BY clauses can be specified in an R/LIST command. If so, the leftmost BY clause becomes the primary sort field. The second BY clause is used to sort data within the groups created by the primary sort field, etc. For example, the command:

```
LIST CUSTOMERS COMPANY CITY ST BY ST BY CITY
```

would produce a report that looks like this:

COMPANY.....	CITY.....	ST
BYTES, INC.	GLENDALE	CA
ARROW CORP	LOS ANGELES	CA
ACME	BELLEVUE	WA
XXX CO.	EVERETT	WA
SMITH CORP.	TACOMA	WA

In this report, all like states (the primary sort field) are grouped together. Within the state groupings, the records appear in city order (the secondary sort field).

Exploding Sorts

If the field being sorted is multivalued, the BY keyword will cause an “exploding sort”. In an exploding sort, each individual value in a multivalued field is treated as if it were in a record by itself. The result is a list or report with an entry for each value in the multivalued field of each record being processed.

Lists or reports created with an exploding sort will have a two-part record key for each record being processed. The first part of the key will be the record's original record key. The second part of the key, separated from the first part by a value mark, will be the value number within the the multivalued field of the sort data.

For example, a CUSTOMER record might have several contact names. If a report were created that sorted by contact name, a portion of the report might look like this:

CUST.NO	COMPANY...	CONTACT....
C100	ACME	DAPHNE
C100	ACME	ROXANNE
C100	ACME	SABRINA
C100	ACME	ZACHARY

Here, the original customer record with four contact names has been “exploded” into four entries on the report.

Correct Use of BY Keyword

LIST CUSTOMERS BY ST

This commands lists records from the CUSTOMERS file sorted into state order.

SELECT CUSTOMERS BY ST BY ZIP BY COMPANY

This command creates an active list of record keys out of records in the CUSTOMERS file. The list is sorted by state and by ZIP code. Within each ZIP code, the list is sorted by COMPANY.

SSELECT CUSTOMERS BY-DSND INVOICE.AMT

This command creates from the CUSTOMERS file an active list of record keys sorted by invoice amount, with the largest invoice amount at the top of the list. Within each like invoice amount, the list is sorted by record key.

COLHEAD

Use COLHEAD to create an alternate column heading for a field.

```
fieldname COLHEAD "col_heading"
```

col_heading

The format override specification *col_heading* contains the alternate text to be used in the column heading. If the column heading created in the format override is longer than the field width, the column heading is truncated to fit the column width.

Correct Use of COLHEAD Keyword

```
LIST INVOICE INV.DT COLHEAD "Inv Date"
```

In this example, the heading "Inv Date" will appear above the column for the field INV.DT. If the display length of the INV.DT field is less than eight characters, the column heading specified in the command will be truncated appropriately.

FOOTING

The FOOTING keyword will cause the R/LIST commands LIST and SORT to produce a running page footing (text at the bottom of each page). The footing can contain text, page numbers, dates, and other information.

```
FOOTING {"text {'options'} "}
```

Using FOOTING

The footing clause consists of the keyword FOOTING followed by the footing specifications in double quotes.

☞ You must use double quotes.

The format of the options for FOOTING are exactly like those of the keyword HEADING. For example:

```
FOOTING "Report produced on 'D' "
```

text

The *text* can consist of any string of characters (including alpha, numeric and punctuation) except double and single quotes.

options

Options can be embedded into the footing text to include page numbers, date, time, and more. Options consist of one or two characters within single quotes. The options available are:

- " QUOTE: Use '' (two single quotes) to embed a single quote into the text of the footing.
- 'B' BREAK: Inserts the value of a breaking field into the heading. It must be the first of the footing options given (after any text) and it must be used in conjunction with the BREAK-ON 'B' option.
- 'D' DATE: Inserts the current date in the DD MMM YY format.
- 'F' FILENAME: Inserts the name of the file that is being listed.

- 'L' NEW LINE: Specifies a line feed.
- 'LL' BLANK LINE: Inserts a blank line (i.e., two line feeds).
- 'N' NOPAGE: Suppresses automatic paging.
- 'P' PAGE: Inserts the current page number.
- 'PP' PAGE JUSTIFY: Right justifies the current page number in a field of four spaces.
- 'Sn' SPACES: Embeds *n* number of spaces into the footing.
- 'T' TIME: Inserts the current time and date.

Correct Use of FOOTING Keyword

`SORT INVOICES BY INV.NO FOOTING "Invoice Listing"`

This is an example of a command using a footing clause. Below are more examples of footing clauses and their output if used with a similar command.

`FOOTING "Invoice Listing for 'D' "`

Invoice Listing for 01 JAN 1987

`FOOTING "Invoice Listing 'S10'Page 'PP' "`

Invoice Listing Page 1

GRAND-TOTAL

GRAND-TOTAL changes the text that accompanies the final total generated by the TOTAL keyword.

`GRAND-TOTAL "text{'options'}"`

Using GRAND-TOTAL

Use the TOTAL keyword to display a total at the bottom of the report (in addition to any subtotals generated). The final total at the end of the report will be marked by the display of three asterisks (***) in the leftmost column of the report. Use the GRAND-TOTAL keyword to change the output from three asterisks to some other text.

text

Substitutes text for the three asterisks normally printed. The text will be truncated to fit into the column width of the leftmost column.

options

The following options enclosed within double quotes are available:

- 'L' LINE: Suppresses the blank line that usually precedes the grand total line. This is useful when it is desirable to condense a report. This option does not override the 'U' option.
 - ☞ GRAND-TOTAL 'L' suppresses a line, while the HEADING 'L' option causes a blank line to be inserted.
- 'P' PAGE: Prints the grand total on a new page.
- 'U' UNDERLINE: Prints a row of equal signs (===) under the grand total.
- 'UU' OVERLINE/UNDERLINE: Prints a row of equal signs (===) over and under the grand total.

Correct Use of GRAND-TOTAL Keyword

LIST CUSTOMERS TOTAL INVOICE.AMOUNT **GRAND-TOTAL "Invoice TOTAL"**
The phrase "Invoice TOTAL" will be printed in the leftmost column of the report at the end, across from the total value for the INVOICE.AMOUNT field. If the leftmost column of the report is narrower than 13 columns, the phrase will be truncated.

HEADING

The HEADING keyword will cause the R/LIST commands LIST and SORT to produce a running page heading. The heading can contain text, page numbers, dates, and other information.

```
HEADING {"text {'options'} "}
```

Using HEADING

The heading clause consists of the keyword HEADING followed by the heading specifications in double quotes.

- ☞ You must use double quotes.

For example:

```
HEADING "Invoices sorted by date'L'Page 'PP'"
```

text

The *text* can consist of any string of characters, including alpha, numeric and punctuation except double and single quotes.

options

Options can be embedded into the heading text to include page numbers, date, time, and more. Options consists of one or two characters within single quotes. The options available are:

- " QUOTE: Use '' (two single quotes) to embed a single quote into the text of the heading.
- 'B' BREAK: Inserts the value of a breaking field into the heading. It must be the first of the heading options given (after any text) and it must be used in conjunction with the BREAK-ON 'B' option.
- 'D' DATE: Inserts the current date in the DD MMM YYYY format.
- 'F' FILENAME: Inserts the name of the file that is being listed.
- 'L' NEW LINE: Specifies a line feed.
- 'LL' BLANK LINE: Inserts a blank line (i.e., two line feeds).
- 'N' NOPAGE: Suppresses automatic paging.
- 'P' PAGE: Inserts the current page number.
- 'PP' PAGE JUSTIFY: Right justifies the current page number in a field of four spaces.
- 'Sn' SPACES: Embeds *n* number of spaces into the heading.
- 'T' TIME: Inserts the current time and date.

Correct Use of HEADING Keyword

SORT INVOICES BY INV.NO HEADING "Invoices Sorted by INV.NO"

This is an example of a command using a heading clause. Below are examples of report headings created by using a heading clause.

HEADING "Invoices Sorted by INV.NO'L'As of 'DL'"

Invoices Sorted by INV.NO

As of 01 JAN 1987

HEADING "Invoices Sorted by INV.NO'S10'Page 'PL'As of 'T'"

Invoices Sorted by INV.NO Page 1

As of 09:00:00 01 JAN 1987

HEADING "Today's report for file 'FL'Page 'PP'"

Today's report for file INVOICES

Page 1

JUSTLEN

Use the keyword JUSTLEN to override the justification or display length of a field. You can override the justification or display length separately, or can combine the override specification in one command.

```
fieldname JUSTLEN [ "just"  
                    length  
                    "justlen" ]
```

Using JUSTLEN

The override specification for JUSTLEN can be an integer value, alpha characters, or a combination of these.

length

An integer alone is interpreted as an override for the display length of the field. The justification remains the same as that specified in the dictionary definition for the field.

just

Alpha characters alone are interpreted as a justification override. The justification specification must be enclosed in quote marks. The display length remains the same as that specified in the dictionary definition for the field. Possible values for justification override are:

- | | |
|---|--------------------------------|
| L | Left justification |
| R | Right justification |
| T | Text (paragraph) justification |
| C | Center justification |

justlen

The combination of text and integer is interpreted as an override for both justification and length. The justification and length specification must be enclosed in quote marks.

Correct Use of JUSTLEN Keyword

LIST CUSTOMER COMPANY **JUSTLEN 40**
This displays the COMPANY field with a length of 40 characters.

LIST CUSTOMER COMPANY **JUSTLEN "R"**
Displays the COMPANY field right-justified.

LIST CUSTOMER COMPANY **JUSTLEN "L40"**
In this example, the COMPANY field is displayed left-justified, 40 characters wide.

LIMIT

The LIMIT keyword restricts the output of data from a multivalued field to those values that meet specified criteria.

```
LIMIT fieldname comparison [value  
                             compare.field]  
  
    (or)  
  
LIMIT fieldname range
```

Using LIMIT

LIMIT is used to “print limit” the values of a multivalued field. LIMIT causes a second-level selection to be executed as each record is displayed, so that only those values in the multivalued field that meet specified criteria are output.

Print limiting is useful when it is desirable to display only a portion of a multivalued field. If the field is simply displayed normally, the entire contents of the multivalued field will be displayed for each record. By specifying LIMIT criteria, however, you can instruct the report to output only those values in the multivalued field that meet criteria you establish.

LIMIT works like a WITH clause, but is applicable only to the display of values. WITH selects records out of the entire file for processing; LIMIT selects values out of one field for display. LIMIT is only applicable to multivalued fields.

comparison

The *comparison* represents the comparison operation that is to take place. Possible values include:

- =, EQ, EQUAL
- >, GT, GREATER THAN
- <, LT, LESS THAN
- >=, GE, GREATER THAN OR EQUAL
- <=, LE, LESS THAN OR EQUAL
- <>, NE, NOT EQUAL

The default comparison is EQUAL. For details on possible values for the compare word, see WITH.

value

value represents the value against which each value in the multivalued field is to be compared. The value should be enclosed in quotes if it is non-numeric. An example of a value comparison is:

```
LIST CUSTOMERS LIMIT INV.DATE > "1-1-86"
```

compare.field

You can also compare *fieldname* with another field in the file. Each value of the multivalued field will be compared against the value in the *compare.field*. For example:

```
LIST CUSTOMERS LIMIT PAID.DATE > DUE.DATE
```

range

The *range* specifies a set of acceptable values that can be met for the data to be displayed. The *range* takes the form:

```
FROM value TO value
```

or

```
BETWEEN value AND value
```

Examples might be:

```
LIST CUSTOMERS LIMIT INVOICE.DATE FROM "1-1-86" TO "12-31-86"
```

```
LIST CUSTOMERS LIMIT INVOICE.AMOUNT BETWEEN 1000 AND 5000
```

Each value in the multivalued field will be compared to the *range*. If the value in the field falls within the range specified by the *range* (inclusive), it will be displayed.

Correct Use of LIMIT

```
LIST CUSTOMERS COMPANY LIMIT INVOICE.AMOUNT >= "1000"
```

This command lists the company name of all customers. In addition, all invoice amounts of \$1,000.00 or greater in the multivalued field INVOICE.AMOUNT are displayed. Invoice amounts of less than \$1,000 are not displayed, even though they may exist.

```
LIST CUSTOMERS COMPANY LIMIT INVOICE.PAID.DT GT INVOICE.DUE.DT
```

This command lists the company name of all customers, and lists all values from the multivalued field INVOICE.PAID.DT in which the date is greater than the contents of the field INVOICE.DUE.DT.

```
LIST CUSTOMERS COMPANY LIMIT INVOICE.DATE FROM "1-1-86" TO ⇨  
"12-31-86"
```

This command lists the company name of all customers, and displays all of the values from the multivalued field INVOICE.DATE in which the value falls into the range of 1-1-86 to 12-31-86 (inclusive).

OCONV

Use the keyword **OCONV** to specify a new or alternate output conversion (output format) for a field. The override specification can be any valid output conversion or formatting specification. Output conversions are most typically used to format fields containing data for money, date, time, or logical (Boolean) values (example: “Y” and “N”).

```
fieldname OCONV "output_format"
```

output_format

Specify a valid output format for *output_format*. For information about available formats, see “Data Formatting” in the chapter “Adding Advanced Features to Windows with Paint” in the section *Developing an Application*, and **OCONV** in the chapter “R/BASIC Command Reference” in *R/BASIC*.

Correct Use of OCONV Keyword

LIST INVOICE INV.DT **OCONV** "D4/"

In this example, the field INV.DT will be displayed with a date conversion that prints four digits for the year, and delimits month, day and year with “/” (slash).

LIST CUSTOMER PHONE **OCONV** "R(###) ###-####"

The phone number will be displayed right justified, with the area code enclosed in parentheses.

ONLY

The **ONLY** keyword may be used with the **LIST** or **SORT** command to cause the report to list only the record keys of the records being displayed. The **TCL** command **LO** is a synonym for “**LIST ONLY**”.

```
LO          filename
LIST ONLY  filename
SORT ONLY  filename
```

Using ONLY

Use **ONLY** when you wish to guarantee that the report will list only record keys. **ONLY** suppresses the default column headings specified in **@CRT** and **@LPTR**. Even if display fields are listed in the command, **ONLY** will suppress the output of all fields except the record key.

ONLY is useful when listing the contents of files that contain data that will not display normally, such as *R/BASIC* object code records.

To change the width and column heading of the ONLY column output, edit the @ID record in the dictionary of *filename*.

☞ ONLY can either precede or follow *filename*.

LO

Use the command LO for the fastest listing of keys to the screen. LO takes no sort or selection criteria.

Correct Use of ONLY keyword

LIST ONLY VOC

Displays only the keys of records in the VOC file.

SORT ONLY CUSTOMERS

Displays only the keys of records in the CUSTOMERS file after sorting the file by record key.

LIST DICT CUSTOMERS ONLY (P)

Displays only the field names of records in the dictionary of the CUSTOMERS file, and prints the report to the printer.

TOTAL

TOTAL is used to generate the sum of a numeric field and display it at the end of the report. If a BREAK-ON has been specified, TOTAL will also display subtotals at each break point.

TOTAL fieldname

Using TOTAL

Use TOTAL to generate totals and subtotals for numeric fields such as dollar amounts. TOTAL will accumulate the value of each field being totalled, and display the result at the end of the report.

If a BREAK-ON has been specified, TOTAL will also output subtotals at each BREAK-ON. Note in the examples that the BREAK-ON and TOTAL fields can be different.

The final total appears at the end of the report, and is identified by three asterisks (***) in the leftmost column of the report. To substitute an option for the three asterisks, see GRAND-TOTAL.

Any number of fields can be totalled in the report. Note that TOTAL can be used with the modifier DET-SUPP (detail suppress) to create reports with *only* total values.

Correct Use of TOTAL Keyword

Sort CUSTOMERS BY ST COMPANY ADDRESS TOTAL BALANCE

This command will generate a report in which the total of the BALANCE field will appear at the bottom of the report.

LIST INVOICES BY COMPANY BREAK-ON COMPANY TOTAL OUTSTANDING.BAL

In this report, a total of the field OUTSTANDING.BAL appears at the bottom. In addition, a subtotal for the field will be generated for each company.

USING

The USING keyword causes the report to use a dictionary other than the dictionary of *filename*.

LIST filename USING alt.filename

USING

USING follows *filename* in the report command. It is followed by the name of the file whose dictionary is to be substituted.

The USING keyword permits you to maintain one master dictionary for several files. This is useful in situations where files have identical layouts, such as master and history files. The master file can have an associated dictionary, and the history file can simply reference that dictionary when reports are generated — no separate dictionary is required for the history file. This minimizes both maintenance and storage requirements.

- ☛ R/LIST will always look first to the dictionary of *filename*. If redirected with the USING keyword, R/LIST will look into the alternate dictionary. In either case, however, R/LIST will always also look in the dictionary of the VOC file for field and keyword definitions. The dictionary of the VOC file can thus serve as a global dictionary for all files.

Correct Use of USING Keyword

LIST INVOICES.HISTORY USING INVOICES

The dictionary of the file INVOICES is substituted for the dictionary of the file INVOICES.HISTORY.

WITH

The filter keyword **WITH** is used with R/LIST commands to generate a subset of records to be processed.

```
with.keyword {EVERY} fieldname [comparison value]
                                MATCHES pattern
                                range
```

Using WITH

Use **WITH** in an R/LIST command when you wish to define which records are to be selected for processing. The **WITH** clause will cause the R/LIST command to examine each record in the file and match it against the criteria following the **WITH** keyword. If the record matches the criteria, it will be added to the list of records to be processed. If the record does not meet the selection criteria, it will be ignored.

WITH does not actually create a file of the subset of records selected. Instead, it creates a list of record keys that represent the records meeting specific criteria.

with.keyword

The *with.keyword* may be **WITH**, **WITHOUT** or any of the following keywords:

WITH

```
BECAUSE {IT HAS}, BECAUSE {ITS}
THAT {HAS}
WHEN {IT HAS}, WHEN {ITS}
IF {IT HAS}, IF {ITS}
WHENEVER {IT HAS}, WHENEVER {ITS}
INCLUDE, INCLUDING
WHICH {HAS}
```

WITHOUT

```
EXCEPT IF {IT HAS}, EXCEPT IF {ITS}
UNLESS {IT HAS}, UNLESS {ITS}
EXCLUDE, EXCLUDING
WITH NO
```

EVERY

Use **EVERY** when *fieldname* is multivalued. Only those records are selected that meet the specified conditions for all values in *fieldname*.

fieldname

The *fieldname* is the name of a field in the file being processed. If *fieldname* is not found in the dictionary of the file being processed, Advanced Revelation will look in the dictionary of the VOC file.

comparison

Use *comparison* to define how WITH compares each record. If *comparison* is not specified, the default value is "=" (equal). Below is a listing of the comparison operators:

=	EQ, EQUAL (TO), THAT IS, LIKE, ARE
<	LT, LESS THAN, BEFORE, UNDER
<=	LE, LESS THAN (OR) EQUAL TO
<>	NE, NOT, NOT EQUAL (TO), DOESNT, ARENT, ISNT
>=	GE, GREATER THAN (OR) EQUAL TO, FROM
>	GT, GREATER THAN, LATER THAN, AFTER, OVER
]	STARTING (IN)
[	ENDING (IN)
[]	CONTAINING MATCH, MATCHES

The comparison operators "[]" (for CONTAINING), "]" (for STARTING WITH), and "[" (for ENDING WITH) can be used in two ways. For example, these two commands are equivalent:

```
LIST CUSTOMERS WITH ANY CONTACTS CONTAINING "SMITH"
```

```
LIST CUSTOMERS WITH CONTACTS [ ] "SMITH"
```

The following example would list invoices with INV.NO beginning with "C":

```
LIST INVOICES WITH INV.NO ] "C"
```

This next example reports the number of customers with company name ending in "INC.":

```
COUNT CUSTOMERS WITH COMPANY [ "INC."
```

These next three commands use comparison words as an actual part of the value. They are equivalent to the previous three examples:

```
LIST CUSTOMERS WITH CONTACTS "[SMITH]"
```

```
SORT INVOICES WITH INV.NO "C]"
```

```
COUNT CUSTOMERS WITH COMPANY "[INC."
```

value

Value is the value that will be compared against the data in the field represented by *fieldname*. The comparison value can be a literal value (text or numeric), or can be the name of another field in the record. If *value* is a literal text value, it should be enclosed in quotes.

MATCHES

If the comparison word **MATCHES** is used, **WITH** will not do a direct comparison against the comparison value. Instead, it will determine whether the data in the field represented by *fieldname* follows the pattern specified in *pattern*.

pattern

When using **MATCHES**, **WITH** matches the data in the field against the mask in *pattern*. The *pattern* should always be in quotes. If the pattern contains literal data, the literal data should be in single quotes, and the entire pattern in double quotes.

Some examples of *patterns* follow:

nA Use *nA* to test for *n* numbers of alphabetical characters.

nN Use *nN* to test for *n* numbers of numerical characters.

nX Use *nX* to test for *n* number of any characters.

nZ Use *nZ* to test for up to *n* number of characters.

"string" Use *"string"* to test for exact text.

☞ Pattern matching is also discussed in the "Creating Windows with Paint" chapter of *Developing Applications*.

range

The *range* is used to specify that the data in the field represented by *fieldname* does not have to meet a specific value. Instead, the data is to fall within a specified set of values.

Ranges for the data values are specified by using the keywords:

FROM ... TO

or

BETWEEN ... AND

The starting comparison value follows **FROM** or **BETWEEN**. The ending comparison value follows **TO** or **AND**.

A *range* selection treats the comparison values as inclusive. In other words, the values used in the comparison are included as part of the range of values to compare against the data in *fieldname*.

Examples of *ranges*:

LIST INVOICES **WITH INV.TOTAL FROM 1000 TO 10000**

LIST INVOICES **WITH INV.TOTAL BETWEEN 1000 AND 10000**

Multiple WITH clauses

Multiple WITH clauses can be specified in a command. The clauses are linked together using the operators AND or OR (an OR is implied if no operator is provided).

Using AND

The effect of using AND between clauses is to narrow the range of records selected. If clauses are connected with AND, the records to be selected must meet the criteria specified in every clause. For example, a command might contain logic like this:

```
command filename WITH clause AND WITH clause
```

For example:

```
LIST CUSTOMERS WITH ST = 'WA' AND WITH CITY = 'SEATTLE'
```

Records to be selected must meet the conditions specified in both WITH clauses.

Using OR

The effect of using OR between clauses is to expand the range of records being selected. Records will be selected if they meet criteria specified in any one of the multiple clauses. The syntax is:

```
command filename WITH clause OR WITH clause
```

For example:

```
LIST CUSTOMERS WITH ST = 'WA' OR WITH ST = 'CA'
```

Records to be selected can meet the conditions specified in either WITH clause.

If the command contains multiple WITH clauses connected with OR, the system can assume certain defaults. If the WITH clauses are all searching the same field and using the same comparison, the WITH clauses can simply “stack up” the comparison values. Advanced Revelation will assume the default OR and fill in the redundant “WITH *fieldname*” portions of the command.

For example, these two commands are equivalent:

```
LIST CUSTOMERS WITH ST = "OR" OR WITH ST = "WA"
```

```
LIST CUSTOMERS WITH ST = "OR" "WA"
```

Using both AND and OR

Multiple WITH clauses can also be connected together with a combination of AND and OR. In this case, the command processes the WITH clauses from left to right, tying together the clauses connected with AND, and separating them from those connected by an OR. In other words, AND has a higher priority than OR.

For example, the command:

```
LIST CUSTOMERS WITH STATUS = 'A' OR WITH STATUS = 'P' ⇨
AND WITH CITY = 'LOS ANGELES'
```

will find all customers whose status is “active” or those whose status is “potential” and who also live in Los Angeles. Customers whose status is “potential” but who do not live in Los Angeles will be ignored.

Each clause separated by an OR is treated as a completely different selection criterion. In the above command, customers will be selected either if they are active, or if they are potential and live in Los Angeles. The clauses linked with an AND are treated as a single unit — each individual clause within the unit connected by AND must be true for the linked clauses as a whole to be true.

Using Parentheses

To change the priority of processing, parentheses can be used to link multiple WITH clauses. For example, the command:

```
LIST CUSTOMERS (WITH STATUS = 'A' OR WITH STATUS = 'P') ⇨
AND WITH CITY = 'LOS ANGELES'
```

will find customers who are either active or potential, and additionally live in Los Angeles. Customers outside Los Angeles, or those who are in Los Angeles but not active or potential, are ignored.

Correct Use of WITH Keyword

```
LIST CUSTOMERS WITHOUT ZIP = 98123
```

List all customers except those in which the ZIP code is 98123.

```
LIST CUSTOMERS WITH ST = 'WA'
```

List all customers in the state of Washington.

```
SELECT CUSTOMERS WITH INV.TOT GE 7000
```

Create an active list of record keys from those customers whose invoice total is greater than or equal to \$7,000.

```
SORT VOC WITH @ID MATCHES "4A"
```

Create a report (sorted by record key) of all records in the VOC file whose record key consists of four alphabetic characters (for example, LIST, SORT, EDIT, etc.).

```
LIST INVOICES WITH CITY BETWEEN "A" AND "D"
```

List invoices for customers whose city falls within the range of “A” to “D” (inclusive — the cities Atlanta and Detroit would be included in the report).

SUM INVOICES PRICE **WITHOUT CITY FROM "A" TO "H"**

Report the sum total of the contents of the PRICE field for those invoices in which the city field does not fall in the range "A" to "H" (inclusive).

COUNT DICT VOC **WHEN @ID CONTAINS "A"**

Report the total number of items in the dictionary of the VOC file in which the record.key (@ID) field has an "A" anywhere within it.

LIST CUSTOMERS **(WITH ST MATCHES '2A')**

List all customers in which the state field contains two alphabetic characters.

Chapter K1

 The SQL Option

Advanced Revelation SQL K 1 . 1

Getting to SQL K 1 . 1





Advanced Revelation SQL

K 1

At Revelation Technologies, SQL is central to our philosophy of integrated data resources: common, transparent ways to manage information across hardware and software boundaries.

Advanced Revelation SQL is a unique integration of an extraordinary application development environment and an industry standard language for database management. The result should be a pleasant surprise.

Advanced Revelation's interactive environment certainly makes using and learning SQL easier than ever before. In addition, the extensive set of tools available give you immediate access to supporting facilities and application environments, while the R/BASIC procedural language offers an efficient, comprehensive programming platform for embedded SQL.

Revelation Technologies' implementation of SQL is exemplary in its adherence to the ANSI standard, so your SQL applications need not be limited by any proprietary constraints. However, if you do intend to use Advanced Revelation SQL as your primary environment, a number of powerful extensions are provided.

The integration of Advanced Revelation and SQL has produced a pre-eminent front-end for database servers. The recently viable database server architecture promises much greater efficiency and performance in processing your transactions. Using an environmental bond, you will find database servers instantly and transparently accessible.

With Advanced Revelation SQL and Environmental Bonding, Revelation Technologies is redefining the scope and character of database management.

Getting to SQL

To begin using SQL, go to the SQL window. There are two ways you can get there:

- Choose the **SQL** option at the Main Menu
- From TCL, enter the command SQL

The SQL Window

The interactive SQL window is the place where you build and execute SQL statements. All of the standard function keys are available to get information, or move easily between environments. The Advanced Revelation Editor is active.

- For a listing of the available edit keys, press [Ctrl-F9] at the SQL window. For a description of the Advanced Revelation Editor, see the chapter "Using the Editor" in *Getting Started*.

SQL

Entering Statements The SQL window currently processes single statements. Carriage returns or line feeds are ignored, so you can format your statement any way you like.

Executing Statements To execute a statement, press [F9].

Correcting Errors If you attempt to execute a statement having incorrect syntax, an error message is displayed and the Editor positions the cursor at the location of the error.

The SQL Assistant

If you are new to SQL, Advanced Revelation gives you an alternative to the frustration of trial and error methods. The SQL Assistant is a dynamic, context sensitive utility to help you construct your statements. When you are in the SQL window, the SQL Assistant is always available. At any point in the construction of your statement, you can activate the SQL Assistant to help you complete the statement. Simply place the cursor where you need help and press [Shift F2]. One of the following actions will then occur:

- If you have not yet entered a command, a popup of the command options is displayed. When you find the command you need press [Enter]. The command is then automatically entered at the SQL window.
- If you have begun a command, a popup will prompt you for the remaining information. Press [F9] to save your entries. This data will then be automatically entered at the SQL window.

When your statement is complete, press [F9] to execute the statement.

The SQL Environment

Default values are provided for all settings defining the SQL environment. For a description of these settings and how to change them, see the chapter "Managing the System".

Chapter K2

An Introduction to SQL

This chapter gives you a brief background of SQL, and discusses the Advanced Revelation SQL environment.

Introduction	K 2 . 1
The Database According to SQL	K 2 . 1
SQL and Advanced Revelation	K 2 . 7



Introduction

SQL is an industry-standard language for database management. It has been widely used on mainframes in the past decade, and as the sharing of data across mainframes, minicomputers and workstations has become more critical, SQL has been recognized as a common link between different types of computers and applications.

Aside from its acceptance in the relational database community, the virtue of SQL lies in its simplicity. There are only a few commands, but sophisticated operations can be achieved by using them in combination.

Using SQL, you can access and modify any SQL database, or create the database yourself. In Advanced Revelation, an SQL window gives you an efficient, interactive environment for doing this. If you are unfamiliar with SQL commands, Advanced Revelation's SQL Assistant can help you construct them.

Programmers can use SQL within R/BASIC, Advanced Revelation's programming language. Embedding SQL commands in this way, enables you to combine information from all of your data resources into central, powerful applications.

For a discussion of embedded or compiled SQL, see the "Programming with SQL" chapter.

The Database According to SQL

From the perspective of SQL, the world is composed of tables: a database you produce is a table or tables, the result of accessing your database is a table, all the information about your database is a table, any system information about you as a user is a table. There is, in short, very little else.

This single method of structuring data leaves you with a gratifying consistency of operation. All you really need to know is what tables are like.

The Elements of a Table

Tables are analogous to files. They are ways of structuring and locating your data. Your SQL database may have one or a number of tables. In order to ensure proper access, each table must have a unique name.

Tables are composed of rows and columns as you might imagine:

CUSTOMERS Table

CUSTOMER_ID	FIRST_NAME	LAST_NAME	ADDRESS
330	Katherine	Carter	821 116th Ave
331	Peter	Hansen	2538 E. Jefferson

Columns The columns of a table describe how the table is to be used by defining the categories and kinds of data that can be entered. In the CUSTOMERS table above, there are four columns, beginning with CUSTOMER.ID and ending with ADDRESS.

Every column within a table must have a unique name, although columns in different tables may have the same name.

☞ See “Naming Names” below for an explanation.

Rows The rows of a table are your data, the information you store in a table. Rows are not stored in any particular order within the table.

In the CUSTOMERS table above, there are only two rows.

Column Values Each row consists of the values you have assigned to the columns in the table. These column values must meet the criteria you establish in your column definitions.

For example, the last row of the CUSTOMERS table above has a value of “Peter” assigned to the FIRST_NAME column.

The Function of SQL

You can use SQL commands to create the tables that will constitute your database, change how the tables are defined, add or remove data, access any specific data stored, and control how the tables are to be used.

There are seventeen commands in Advanced Revelation SQL, and all of these can be usefully categorized as one of the following types:

- Data Definition Language
- Data Manipulation Language
- Data Control Language

The table below lists the commands by functional category:

*Table K2-1***Types of Advanced Revelation SQL Statements**

Category	Command
Data Definition	CREATE TABLE CREATE VIEW DROP TABLE DROP VIEW DECLARE
Data Manipulation	INSERT UPDATE DELETE SELECT FETCH ALTER TABLE
Data Control	BEGIN TRANSACTION COMMIT ROLLBACK SET TRANSACTION OPEN CLOSE

K 2**An Introduction to SQL****Forming SQL Statements**

You will find that SQL statements fit together in familiar and comfortable ways. For example, to create a table of your customers you could use the statement:

```
CREATE TABLE CUSTOMERS CUSTOMER_ID INTEGER
```

You would then have a table consisting of the single column CUSTOMER_ID. Only integer numeric data is acceptable for this column.

As another example, you might check for the first name of one of your customers with the statement:

```
SELECT FIRST_NAME FROM CUSTOMERS  
WHERE LAST_NAME = "Hansen"
```

The Parts of an SQL Statement

Statements in SQL are built using a basic set of elements or constructs. You will not need to use all of these elements for every statement.

Commands

The command is always the first element in your SQL statements. In the examples above, the CREATE TABLE and SELECT commands were used.

Column and Table Names

In all of your SQL statements column and table names will be your terms of reference, whether you are accessing data or creating a table.

In the example above, the CUSTOMER_ID column and the CUSTOMERS table were used.

Keywords

There are quite a few keywords in SQL. Some of them exist solely to make the statement easier and more natural to use, others are very powerful.

In the above examples, **FROM** and **WHERE** are keywords.

For a complete listing and description of keywords, see “SQL Keywords” in the “SQL Reference” chapter.

Operators

There are three different types of SQL operators:

Math	+, -, *, /
------	------------

Comparison	=, <, >, <=, >=
------------	-----------------

Boolean AND, OR, NOT

Literal Data

Literal data is always enclosed in quotes, either double or single quotes. You may use literal data most often when you enter data into your tables, but you can also use literals in comparisons.

Using Parentheses

Parentheses are used in SQL much as they are in algebraic equations, enclosing:

- Column definitions or assignments, such as:

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID
INTEGER PRIMARY KEY, FIRST_NAME CHAR(15),
LAST NAME CHAR(25), ADDRESS CHAR(35), CITY
CHAR(25), STATE CHAR(2))
```

- Equations involving columns, such as:

```
SELECT DESCRIPTION, (QUANTITY * PRICE)
FROM PRODUCTS, INVOICES
```

- A parameter to a function keyword, such as:

```
SELECT AVG (PRICE) FROM PRODUCTS
```

- Nested queries, such as:

```
SELECT CUSTOMER_ID FROM CUSTOMERS
WHERE PRODUCTS.PRODUCT CODE =
  (SELECT CODE FROM INVOICES
   WHERE CODE = 010120)
```

- Associated UNION statements, such as:

```
SELECT * FROM PRODUCTS__88
UNION
(SELECT * FROM PRODUCTS_87
UNION
SELECT * FROM PRODUCTS)
```

When parentheses are required, it will be clearly stated in the documentation for the statement. The SQL Assistant will also help with missing parentheses.

Naming Names

Identifiers, names for columns, tables, and views, are all uppercase in SQL. The first character must be an alpha character (A-Z), but any of the following characters can be numeric or the underscore character (_).

Advanced Revelation SQL places no limit on the length of identifiers, although standard SQL restricts these to eighteen characters.

- ☞ If you may want at some point, to move your SQL applications to another environment, it is best to enforce the eighteen character limit for ease of portability.

Using the Full Name for a Column

In SQL, the full name for a column consists of the column name preceded by a period and the name of the table. For example, in a CUSTOMERS table, the full name for a CUSTOMER_ID column would be:

```
CUSTOMERS.CUSTOMER_ID
```

Full column names guarantee that there is no ambiguity in your reference. Although each column within a table must have a unique name, as must each table within the database, there may be columns of the same name in other tables. Prefixing the column name with the table name resolves any potential conflicts.

- ☞ When 'joining' tables for example, use of full column names may be necessary. For more information see "Joining Tables" in the chapter "Querying the Database".

User Names and Tables

Advanced Revelation SQL prefixes all tables with the name of the user that created the table. In order to access a table created by another user, you will need to specify the user name prefix for the table.

For example, a reference to an INVOICES table created by a user named Annette would be:

```
ANNETTE . INVOICES
```

If you needed data from the QUANTITY column in this table, your reference would be:

```
ANNETTE . INVOICES . QUANTITY
```

To access a user table outside of the SQL environment, prefix the reference with @ (at-sign).

- ✎ For more information about accounts, see the chapter “Users and Accounts: Planning Access and Security” in *Developing an Application*.

Tables and Transactions

When you update a table, SQL gives you a choice of how you want your statements to be executed. You can have each statement executed immediately as an individual operation, or you can ‘package’ the operations in a *transaction* and have them executed as a unit.

A transaction might be a single SQL statement or a sequence of statements. The purpose of this *transaction processing* provided by SQL is to ensure that either all of the operations comprising a transaction are executed, or none of the operations are executed. It is up to you to determine the sequence of statements that constitutes a transaction.

How Transactions are Processed

Transaction processing means that your update operations are not made directly. As a matter of fact, the operations may not be executed at all, if this is what you wish. A transaction is first written to an intermediate file, a kind of waiting room for transactions. The changes do not really occur until:

- You decide that you do want the transaction to be executed, (*committed*) or not (*rolled back*). This also defines the operations constituting a transaction.
- Any constraints imposed by your system security are satisfied

Because your transactions are collected until you want them to be processed, you are able to group dependant or related operations and have them processed as a unit.

This kind of control can be critical for applications, such as financing or accounting packages, that rely upon interdependent operations.

Should SQL fail to complete the execution of a transaction, due to a system error or failure of some type, you are notified when you log back in to Advanced Revelation. You then have the option of completing the transaction.

For more information about transaction processing, see the chapter “Managing Files and Records” in *Filing Systems*.

SQL and Advanced Revelation

SQL and Advanced Revelation employ compatible philosophies. The relational model is at the heart of both. However, as is the case with any two relational database designs, each implements the model somewhat differently.

SQL for Advanced Revelation has been fully integrated into the Advanced Revelation system. You can:

- Use any of the familiar Advanced Revelation tools with SQL
- Display the interactive SQL window or use SQL commands at the TCL window from anywhere within Advanced Revelation

On the other hand, the differences in the way Advanced Revelation and SQL work mean you need to:

- Account for data types and fixed-length data structures
- Use a slightly different terminology
- Use different naming conventions for constructs
- Accept standard data validation and conversion methods
- Understand the handling of null values
- Distinguish data files from other types of files
- Understand the way changes to data are handled

The issues and differences to account for when using SQL are largely a product of its mainframe heritage. The migration of SQL to microcomputer workstations has provided SQL with refined user interfaces, such as Advanced Revelation offers. However, in maintaining its common value across architectures, the character of SQL may prove more rigid than you are accustomed.

The ANSI Standard

Implementations of SQL itself are often quite different from each other as well. Although an ANSI (American National Standards Institute) standard does exist, not all SQL products adhere to it as rigorously as Advanced Revelation.

Advanced Revelation SQL is a superset of ANSI standard SQL. The majority of the standard SQL commands and operations are provided, as well as some important extensions.

- ☛ Currently, the SQL security operations GRANT and REVOKE are not available in Advanced Revelation SQL. For more information about Advanced Revelation extensions, see “Extensions to SQL” in the “SQL Reference” chapter.

Coming to Terms with SQL

Advanced Revelation describes a database as a hierarchy of files, records, fields, and values. These are common and acceptable terms in the industry, but they are not standard usage for SQL. Instead, SQL deals in tables, rows, and columns. The table below illustrates the corresponding terms.

Advanced Revelation	SQL
account	database
volume	<i>n/a</i>
file	table
record	row
field	column
value	column value

Notice that SQL does not have a term corresponding to a volume. Advanced Revelation includes specifications for a volume as an option with the CREATE TABLE commands.

The concept of an account in Advanced Revelation is broadly analogous to an SQL database. Although it would be possible for you to have more than one database in an account, for reasons of maintenance and security, it is strongly recommended that you do not do so.

Assigning Multiword Identifiers

Advanced Revelation frequently uses a period to delimit identifying names of more than one word, such as CUSTOMER.ID or FIRST.NAME. These might be the names of volumes, fields, files, or any other construct. SQL requires the use of the underscore character (`_`) for the same purpose. Consequently:

- Always use the underscore character (`_`) as a delimiter in SQL

We suggest that you use this same convention with all your applications, although Advanced Revelation will not require you to do so.

Here are some examples of how compound names might be used in SQL:

```
LAST_NAME
CUSTOMER_ID
```

When full column names are used to specify the CUSTOMERS and INVOICES tables, the above examples would appear as:

```
CUSTOMERS.LAST_NAME
INVOICES.CUSTOMER_ID
```

Accessing Files with SQL

Whether or not you choose to use tables exclusively, Advanced Revelation SQL does not constrain you to do so. You can access any Advanced Revelation or foreign file that is attached.

- ☛ See the ATTACH command in the “TCL Command Reference” chapter of *TCL*. For information about volumes, see the chapter “Advanced Revelation Volumes” in *Filing Systems*.

Simply use the name of the file and field in place of a table and column reference in your SQL statements. For example, to select the names from an OLD_CUSTOMERS file:

```
SELECT LAST_NAME, FIRST_NAME FROM OLD_CUSTOMERS
```

- ☛ If the identifiers use a period (.) rather than an underscore (_) as a delimiter (L.NAME and F.NAME), SQL will interpret the references properly. Do not attempt to use a period as a delimiter. This is an SQL syntax error.

Using Columns with Multiple Values

The Advanced Revelation architecture includes the capability to assign multiple values to a column. These *multivalued* columns add another ‘repeating’ row for each value in the column. The rows are repetitive because they duplicate the values for all columns that are not multivalued.

For example, a selection from the INVOICES table with the multivalued column QUANTITY would appear as follows:

CUSTOMER_ID	DATE	QUANTITY
349	5/9/89	2
349	5/9/89	1
349	5/9/89	1
349	5/9/89	1

The values are repeated for the non-multivalued INVOICE_NO and DATE columns. The number of repeating rows corresponds to the number of values in the largest multivalued column.

- ☛ Advanced Revelation SQL does not actually store the repeating rows. Instead, the row is stored only once and dynamically 'expanded' into the additional rows whenever you select the row. For information about defining multivalued columns, see the REPEATING keyword in the "SQL Reference" chapter.

The KEYQUAL Column When you create a table with a multivalued column or columns, an additional key column is automatically defined for the table. This is the KEYQUAL column, so named because it qualifies the primary key for the table by uniquely identifying each value in the multivalued columns.

The KEYQUAL column gives you a consistent way to access the values in a multivalued column.

- ☛ Like a primary key column, the KEYQUAL column cannot be modified. See PRIMARY KEY and KEYQUAL in the "SQL Keywords" part of the "SQL Reference" chapter.

For example, using the INVOICES table illustrated above, you might select the second value in the QUANTITY column:

```
SELECT QUANTITY FROM INVOICES
WHERE CUSTOMER_ID = 349 AND KEYQUAL = 2
```

Null Handling

A null value signifies that no data exists. The Advanced Revelation architecture interprets null fields as zero for arithmetic operations, and as False for comparisons. In SQL, a null value is interpreted as neither zero nor False, but as unknown. It is important that you keep these differences in mind when moving between the environments.

The SQL method of null handling affects:

- Arithmetic operations
- Comparison operations
- Boolean operations
- Ordering precedence

Arithmetic Operations An arithmetic operation (+, -, *, /) involving a null value will evaluate to null, since at least one of the elements in the operation is unknown.

Comparison Operations Any two null values are not considered equal. In addition, in WHERE or HAVING clauses, a comparison involving a null value will evaluate to null, since at least one of the elements in the operation is unknown.

- ☞ SQL provides the NULL keyword to test for null values. See the "SQL Keywords" part of the "SQL Reference" chapter.

Boolean Operations

Because null represents an unknown value, SQL logic must account for True, False, and unknown, rather than simply True and False. The following truth tables illustrate this (where ? indicates unknown):

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

OR	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

NOT	
T	F
?	?
F	T

Ordering Precedence

When ordering values, SQL places null values first in the order of precedence.

- ☞ See the ORDER BY keyword in the "SQL Reference" chapter.

This conflicts with Advanced Revelation's method of placing null values last in the order.



Chapter K3

 Building a Database

This chapter describes how to define the tables and views that form your database.

Creating Tables K 3 . 1

Viewing Information K 3 . 8





Creating Tables

When you create a table, you are designating a location for data. You are also configuring the table to accept and use data in particular ways. The number and size of tables comprising a particular database, the relationships, and the column definitions all have a lot to do with an efficient database design.

A table is defined by its columns. You can create tables of one or any number of columns. Each column within a table must have a unique name and accept a particular type of data.

To create a table, use the CREATE TABLE command. The basic format is:

```
CREATE TABLE tablename column_name column_definition
```

where *tablename* is the identifying name for the table, and *column_name* is the identifying name for a column you are defining for the table. The *column_definition* defines the kind of data that can be assigned to the column. For example, the statement

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER KEY)
```

creates a table named CUSTOMERS. The CUSTOMERS table has, at present, a single column named CUSTOMER_ID. The column is additionally defined by the keywords INTEGER and KEY. The data type keyword INTEGER defines the acceptable data for this column. KEY defines the column as the primary key column for the table (see “Assigning Primary Keys” below).

Defining a Column

The only information required to define a column in SQL is the name of the column and the type of data accepted.

- ✎ The PRIMARY KEY keyword, however, must be used to define at least one column in every table.

You can also include additional definitions for a column. Advanced Revelation SQL requires that you maintain a specific order when using column definitions.

The keyword order is as follows:

1. Data type keyword
2. DEFAULT *value*
3. NOT NULL
4. UNIQUE
5. PRIMARY KEY
6. REFERENCES *tables*
7. REPEATING

Specifying a Data Type

Although there are a number of different data types you can specify, all refer either to characters, numbers, dates, or logical values. The data type specification always immediately follows the name of the column you are defining.

Two of the keywords Advanced Revelation SQL provides for specifying character data types are:

- **CHAR(*n*)**
For fixed-length data, where *n* is the maximum length of the data
- **VARCHAR**
For variable-length data

For example, the statement

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY, LAST_NAME CHAR(25))
```

creates a table **CUSTOMERS** having a **CUSTOMER_ID** and a **LAST_NAME** column. The **LAST_NAME** column would accept up to 25 alphanumeric characters.

Two of the keywords you can use to specify numeric data types are:

- **INTEGER**
For integers
- **DECIMAL**
For fixed-point numbers

For example, the **CLIENT** column in the statement

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY)
```

will require integer data.

- ☞ For a listing of available data types, see “SQL Data Types” in the “SQL Reference” chapter, and the “Advanced Revelation Files and Records” chapter of *Filing Systems*.

Setting a Default Value

If a column is likely to have a standard value for all rows, you can assign a default value to the column. To do this, use the **DEFAULT** keyword followed by the value you wish to assign to the column.

- ☞ The value you assign must conform to the data type definition for the column.

For example, the statement,

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY, ST CHAR(2) DEFAULT "WA")
```

will assign a default value of "WA" to the ST column in every row.

The format is:

```
DEFAULT value
```

where *value* is one of the following:

Literal Data A literal can be any data enclosed in single or double quotes.

The NULL Keyword The NULL keyword assigns a null value to the column.

☞ The column must be defined to accept a null value. See "Requiring Data" below.

The CURRENT Keyword The CURRENT keyword assigns the current date and time to the column.

The USER Keyword The USER keyword assigns the name of the current user to the column.

Requiring Data A column without any data has a value of null. You may want to be certain however, that the column receives data.

You can ensure that a column must have data entered by using the NOT NULL keyword. For example,

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY, LAST_NAME CHAR(25) NOT NULL)
```

will not allow any of the values in the LAST_NAME column to be null (empty).

In insert or update operations (the INSERT and UPDATE commands), an SQL error message will be displayed if the user does not enter data for a NOT NULL column.

Requiring Unique Data If you need to guard against duplicate values in a column, use the UNIQUE keyword. SQL requires that UNIQUE is always preceded by NOT NULL.

It is not necessary to specify NOT NULL and UNIQUE for primary keys. This is implicit in the definition. See "Assigning Primary Keys" below.

An example might be:

```
CREATE TABLE PRODUCTS (PRODUCT_CODE INTEGER
KEY, DESCRIPTION VARCHAR NOT NULL UNIQUE)
```

Any data entered in the DESCRIPTION column will then be checked against all other values in the column. If any duplicate values are found, SQL displays an error message.

You can also apply UNIQUE to a list of columns, rather than including UNIQUE in the definition for each column. To do so, include UNIQUE only in the definition of the last column that you wish to be unique. The columns are enclosed in parentheses and separated by commas.

- ☞ You still need to include NOT NULL with each column definition.

For example, the statement

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY, ADDRESS CHAR(25) NOT NULL, PHONE
CHAR(12) NOT NULL UNIQUE (ADDRESS, PHONE))
```

defines both the ADDRESS and PHONE columns as requiring unique data.

Assigning Primary Keys

A primary key is a column used as the primary means of identifying a row. This guarantees that your references to rows are unambiguous.

- ☞ Advanced Revelation SQL requires that each table have a primary key column. An error message is displayed if you attempt to create a table without defining a primary key.

To assign a primary key, use the PRIMARY KEY or KEY keyword.

- ☞ KEY is an optional abbreviation for the keyword.

For example:

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY)
```

You can specify one or more columns to be used as the primary key for a row. The previous example might be extended to:

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER
KEY, ADDRESS VARCHAR KEY)
```


Because the values in primary key columns uniquely identify each row in the table, once the table is created, you cannot modify your definition for a primary key column. This helps maintain the integrity of your references.

When you define a column as a primary key, it is not necessary to specify NOT NULL and UNIQUE. This is implicit in the definition.

Referencing Foreign Keys

A foreign key is a column that references a primary key column in another table. The column values for the foreign key column should correspond to the column values for the referenced key column. When you need to join tables, a foreign key is your hook into another table.

☛ For more information about joins, see “Joining Tables” in the chapter “Querying the Database”.

For example, if the primary key of a CUSTOMERS table is the CUSTOMER_ID column, and the CUSTOMER_ID column in an INVOICES table is a foreign key referencing the CUSTOMER_ID column, the query:

```
SELECT LAST_NAME, SUM(QUANTITY * PRICE) FROM
CUSTOMERS, INVOICES
WHERE CUSTOMERS.CUSTOMER_ID =
INVOICES.CUSTOMER_ID
```

would find the corresponding account balance for each customer. The last name and balance are properly related because the values of the CUSTOMER_ID column in the INVOICES table are identical to the values in the CUSTOMER_ID column of the CUSTOMERS table.

☛ Note that the column name for the foreign key need not be the same as the referenced column. The CUSTOMER_ID column in the INVOICES table might instead be named INVOICE_NO. It is the values that must be identical.

The REFERENCES keyword gives you a way to document foreign key references for a column. REFERENCES is followed by a list of the table or tables in which the defined column is a foreign key. For example, the definition for the CUSTOMER_ID column might be:

```
CREATE TABLE CUSTOMERS CUSTOMER_ID INTEGER
REFERENCES INVOICES
```

Using REFERENCES, you can help track the relationships you establish between your tables.

Allowing Multiple Values

In Advanced Revelation SQL, you can define a column to accept more than one value. These *multivalued* columns are unique to Advanced Revelation, and allow you to fully access any Advanced Revelation file that uses multivalued fields.

- ✎ Standard SQL does not support multivalued columns. Do not use this facility if you intend to port your application to another SQL environment.

To define a multivalued column, use the **REPEATING** keyword. For example, the **QUANTITY** column in

```
CREATE TABLE INVOICES (CUSTOMER ID INTEGER
KEY, CODE INTEGER, QUANTITY INTEGER
REPEATING)
```

will accept more than one value. The maximum number of values you can enter is limited only by the record size.

Designating a Volume

Advanced Revelation uses the concept of 'volumes' to group the information related to an application. Whenever you create a table, it is always assigned to a volume.

You have the option of designating a volume when you create the table. If you decide not to specify a volume, the table is assigned to the default volume **DATAVOL**. This is the volume where all of your files and tables will exist, unless you decide otherwise.

- ✎ There are very good reasons for building your applications in separate volumes. See the "Advanced Revelation Volumes" chapter in *Filing Systems*.

You can specify a volume only with the **CREATE TABLE** command. The volume identifier is always preceded by the keyword **ON**. For example,

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER KEY) ON SALES
```

assigns the **CUSTOMERS** table to the Advanced Revelation volume **SALES**.

- ✎ To establish a volume, see the TCL command **SETVOLUME** in the chapter "TCL Command Reference" of *TCL*.

Defining Multiple Columns

Usually you will be defining more than one column at a time in your **CREATE TABLE** statements. To do so, separate each column definition with a comma, and enclose all of the columns defined in parentheses. For example, the statement

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER KEY, LAST_NAME
CHAR(25), FIRST_NAME CHAR(15), ADDRESS VARCHAR, CITY
CHAR(25), ST CHAR(2), ZIP INTEGER)
```

defines the columns `CUSTOMER_ID`, `LAST_NAME`, `FIRST_NAME`, `ADDRESS`, `CITY`, `ST` and `ZIP`. The commas delimit each column definition.

Altering a Table Definition

After you have created a table, and before any data is entered into the table, you can change it. However, although changing a table may be a simple operation for the user, the system may be required to rebuild the complete table. This can slow down processing and impact available memory. For this reason, you should avoid altering your tables except when necessary.

- ✦ Although you can modify Advanced Revelation tables or files, bonded environments may be more restrictive. Refer to the documentation for your bond. For more information see the “Introduction to Bonding” chapter of *Environmental Bonding*.

To alter a table, use the `ALTER TABLE` command. You change a table by changing the columns within the table. The requirements and options for defining a column are exactly the same as for the `CREATE TABLE` command.

The `ALTER TABLE` command gives you three ways to redefine a table.

Adding a Column You can add a new column to a table using the `ADD` keyword. The keyword is followed by the name of the column to add and the column definition. For example, the statement

```
ALTER TABLE CUSTOMERS ADD PHONE CHAR(12)
```

will add a `PHONE` column to the `CUSTOMERS` table. You can only add one column at a time.

Modifying a Column The `MODIFY` keyword enables you to change the definition for an existing column. The keyword is followed by the name of the column and the new definition:

```
ALTER TABLE CUSTOMERS MODIFY ADDRESS CHAR(25)
```

The `ADDRESS` column will now require a maximum of 25 characters.

Deleting a Column To delete a column, use the keyword `DROP`. The keyword is followed by the name of the column to delete. For example, the statement

```
ALTER TABLE CUSTOMERS DROP PHONE
```

will remove the `PHONE` column from the `CUSTOMERS` table.

Viewing Information

To refer to data in SQL is to refer to a table. The table may be a complete table, part of a table, or a combination of more than one table.

The tables that you build with the **CREATE TABLE** command allocate storage space for data. For this reason these are referred to as 'base' tables. You can also build tables which are simply definitions of a table: 'virtual' tables, that do not have any storage allocated, but receive their data from base tables. These are called 'views'.

The Purpose of Views

You can use views to define custom tables, collect specific information from a number of tables, or restrict access to particular data. Some situations for which you might want to create a view are:

- You have a fairly large table, but most of the time you only access certain columns.
- You regularly access related information located in a number of tables.
- Security issues require that the information should not be equally accessible to all users.

In addition to deriving their information from base tables, views may also consist of data from other views, or from a combination or series of base tables and views.

Although views do not occupy any storage space, they do assess a cost in time. Since a view is simply a definition, each time it is accessed the view must get its information from other tables or (circuitously) other views. This may take slightly longer than directly accessing a comparable base table.

Creating Views

To create a view, use the **CREATE VIEW** command. A **CREATE VIEW** statement consists of the names of the columns to be defined for the view, and a query.

The query determines the scope of the view, the amount and types of information that the view will contain. For example, you might define a view of a **CUSTOMERS** table by:

```
CREATE VIEW CUSTOMER_CONTACT (CUSTOMER_ID, F_NAME, L_NAME,  
PHONE) AS  
  SELECT CUSTOMER_ID, FIRST_NAME, LAST_NAME, PHONE FROM  
    CUSTOMERS  
  WHERE ST = "WA"
```

The **CUSTOMER_CONTACT** view would consist of the **CUSTOMER_ID**, **FIRST_NAME**, **LAST_NAME** and **PHONE** columns as defined in the **CUSTOMERS** table. Notice that the column names for the view may or may not duplicate the column names for the base table.

You need not specify the column names for the view if you want to use the names as they appear in the base table. The previous CREATE VIEW statement could be changed to:

```
CREATE VIEW CUSTOMER_CONTACT AS
  SELECT FIRST_NAME, LAST_NAME, PHONE FROM CUSTOMERS
  WHERE ST = "WA"
```

Accessing Views

Having created a view, you can then query the view at any time, just as you would a base table:

```
SELECT PHONE FROM CUSTOMER_CONTACT
  WHERE LAST_NAME = "Robbins"
```

The query returns the phone number of the customer to contact.

- ☛ Because a view is a definition, rather than a location of data, you cannot access views outside of the SQL environment.



Chapter K4

 Querying the Database

This chapter describes how to find and retrieve information from your database.

A Query Language K 4 . 1

Forming Queries K 4 . 1





A Query Language

SQL means Structured Query Language. Its primary use is to “ask” the database for information. A query searches tables row-by-row for the appropriate data.

Because queries select rows, and rows consist only of columns, you never explicitly refer to a row: the terms of interaction for your queries are table names and column names. The result of any successful query is always a table of a least one row and column.

- ☛ The table returned by a query is known as a “results” table. A results table is different from any other table only in that you can return to it only by repeating the query: its existence is limited to the duration of the query’s result.

For example, you might have a database containing a table of product information. The table may be partially illustrated as below:

PRODUCTS

PRODUCT_CODE	DESCRIPTION	PRICE
010100	RED SAILBOAT TIE	25.95
010120	TEAL PAISLEY TIE	25.95
010210	BLACK LEATHER GLOVES, MEN'S	30.00
020120	ROYAL SILK SCARF	35.90
030460	MULTI SILK SWEATER	104.00

- ☛ The PRODUCTS table is used in the Advanced Revelation Tutorial, and is stored in the TUTOR volume.

Suppose you wanted to find out the price for the red sailboat tie. You would check the DESCRIPTION column and follow that row across to the number in the PRICE column.

Since this is SQL, your query would return a results table of a single row and column consisting of the price for the tie. If any of the other rows also had a DESCRIPTION value of “RED SAILBOAT TIE”, prices for these rows would be selected as well.

Forming Queries

Queries use the SELECT command. To formulate your queries you need to know:

- The name of the table(s) to query
- The name of the column(s) within the table(s) to reference
- Any conditions to describe the data required

Here is a basic format for your queries:

```
SELECT column FROM table
```

Selecting All Columns

You can select every column for every row in the table. To do this, use the * (asterisk) operator instead of a column name. For example,

```
SELECT * FROM PRODUCTS
```

will return a results table of all the columns and rows in the PRODUCTS table, essentially a copy of the table.

- ✎ The KEYQUAL column, an additional column automatically defined for multivalued columns, does not appear in SELECT * operations. For more information, see "Allowing Multiple Values" in the chapter "Building a Database" and "Selecting Values from Multivalued Columns" at the end of this chapter.

Selecting a Specific Column

By specifying a column name, you can select that column for every row in the table. For example, the query

```
SELECT PRICE FROM PRODUCTS
```

will return a results table consisting of a PRICE column value for every row in the table.

Selecting Multiple Columns

To select more than one column, list the column names delimited by commas. For example, the query

```
SELECT DESCRIPTION, PRODUCT_CODE FROM PRODUCTS
```

will return a results table consisting of the DESCRIPTION and PRODUCT_CODE column values for every row in the table.

Applying Functions to Columns

SQL supplies six function keywords that you can apply to columns. These are:

```
COUNT  
COUNT(*)  
AVG  
SUM  
MAX  
MIN
```

The functions are followed by the name of a column (or an equation using columns) in parentheses.

Counting Distinct Rows

Using the **COUNT** keyword, you can count the number of rows in a table. When you use **COUNT** with a column, any rows having a null value for that column will not be counted.

For example,

```
SELECT COUNT(DISTINCT PRODUCT_CODE) FROM
PRODUCTS
```

will count the rows in **PRODUCTS** having a non-null value for the **PRODUCT_CODE** column. Duplicate rows are not included in the count.

All rows are evaluated unless you specify the **DISTINCT** keyword.

Counting All Rows

To count all rows in the table regardless of null values, use the * (asterisk) operator instead of a column name. Any duplicate rows are included in the count.

✎ Use of the **DISTINCT** keyword is not allowed for **COUNT(*)**.

For example,

```
SELECT COUNT(*) FROM PRODUCTS
```

will count all rows, whether null or duplicates, in the **PRODUCTS** table.

Averaging a Column

You can average the values for a numeric type column with the **AVG** keyword. For example, to get an average cost for all of the different types of products in stock, use the query:

```
SELECT AVG(PRICE) FROM PRODUCTS
```

All rows are evaluated unless you specify the **DISTINCT** keyword. However, in either case, null values are eliminated before the calculation.

Summing a Column

To sum or total the values for a column, use the **SUM** keyword. For example, the query

```
SELECT SUM(PRICE * QUANTITY) FROM PRODUCTS
```

will return the total retail value for the product inventory.

All rows are evaluated unless you specify the **DISTINCT** keyword. However, in either case, null values are eliminated before the calculation.

Finding Maximum and Minimum Values

You can select a maximum or minimum value from a column with the **MAX** or **MIN** keywords. For example, to find both the maximum and minimum prices charged, use the queries:

```
SELECT MAX(PRICE) FROM PRODUCTS
```

```
SELECT MIN(PRICE) FROM PRODUCTS
```

Calculating with Columns

To perform calculations with columns, use math operators and column names to return a value. You can combine and order your column equations using parentheses, just as in algebraic equations.

The math operators available in SQL are:

+	Addition
-	Subtraction
*	Multiplication
/	Division

For example, to find the tax on a particular product, use the query:

```
SELECT DESCRIPTION, (PRICE * .85) FROM PRODUCTS
```

This will return a results table consisting of the **DESCRIPTION** column and a column of taxed amounts for every row.

You can also use column equations with a function keyword. For example, to get a total wholesale value for the products, you might use the query:

```
SELECT SUM((PRICE * QUANTITY) * .35) FROM PRODUCTS
```

Setting Conditions

You will often be interested in searching a subset of a table for information: a limited number of rows and columns, rather than a complete table. To establish the limits for your query, use the **WHERE** keyword followed by the conditions for selecting the rows (a **WHERE** "clause").

The SQL query corresponding to the initial example for the **PRODUCTS** table would be:

```
SELECT PRICE FROM PRODUCTS  
WHERE DESCRIPTION = "RED SAILBOAT TIE"
```

The **SELECT** operation will first apply any conditions you might specify. In this case, only rows in which the **DESCRIPTION** column has the value **RED SAILBOAT TIE** are selected. The results table returned would be:

```
010100
```

A **WHERE** clause tests each row in the table to see if a column meets the specified conditions. The conditions clause is very flexible, giving you a variety of ways to limit the selection and test data. The format of a **WHERE** clause is:

```
WHERE expression comparison expression
                        subquery
```

The *expressions* might be literal values, columns or column equations. The *comparison* could be one of SQL's comparison operators described below.

- ✎ See "SQL Keywords" in the "SQL Reference" chapter for a complete description of the **WHERE** clause.

Making Comparisons You can compare column values, literal values, or a mix of both. SQL gives you the standard mathematical operators to use for comparisons:

=	Equals
<>	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Among the keywords you can use for comparisons are:

```
BETWEEN
IS NULL
IN
LIKE
```

Using the Comparison Operators You can select a range of values using a **WHERE** clause. To do this, use one of the comparison operators:

For example, the query

```
SELECT COUNT(PRICE) FROM PRODUCTS
WHERE PRICE < 32.50
```

will count only those **PRICE** columns having a value less than \$32.50.

The query

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE >= 100.00
```

will return a results table consisting of the **DESCRIPTION** column for each row having a **PRICE** value greater than or equal to \$100.00.

*Using Comparison
Keywords*

The comparison keywords are very English-like in their usage. All of these keywords can be prefixed with NOT for a negative comparison.

Testing for a Range. The BETWEEN keyword enables you to test for a range of values based on a specified low and high value.

☛ You must specify the low value first.

For example,

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE BETWEEN 26.00 AND 32.50
```

The results table will consist of the DESCRIPTION column for each row having a PRICE value between \$26.00 and \$32.50.

You could change this example to test for all PRICE values not between \$26.00 and \$32.50:

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE NOT BETWEEN 26.00 AND 32.50
```

Testing for Null. To test for null (empty) columns, use the IS NULL keyword:

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE IS NULL
```

The results table returned will consist of the DESCRIPTION column for each row having a null PRICE value.

Using NOT, you can reverse this example to find all handle types having a non-null PRICE value (although this might be zero):

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE IS NOT NULL
```

Testing for Inclusion. You can compare a value for inclusion within a list of values with the IN keyword. IN is followed by the list of values in parentheses and delimited by commas:

```
SELECT PRICE FROM PRODUCTS
WHERE DESCRIPTION IN ("GREEN SAILBOAT TIE",
"ROYAL SILK SCARF",
"BROWN LEATHER GLOVES, MEN'S")
```

Testing for Similarity. To make a comparison based on similar values, use the LIKE keyword. SQL uses % (percent) and _ (underscore) as special “wildcard” characters for LIKE comparisons:

%	Matches any combination of characters
_	Matches any single character

For example,

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
WHERE DESCRIPTION LIKE "MULTI
SILK _ _ _ _ _ _ _"
```

will test for a DESCRIPTION value equal to "MULTI SILK" plus seven characters. The query will return a results table of a DESCRIPTION and PRICE column for each row having a matching DESCRIPTION.

The query,

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
WHERE DESCRIPTION LIKE "M%"
```

would return the same results table as the previous example. The LIKE comparison will match any handle types that begin with “M”. There may be any number of subsequent characters.

For more information about any of the comparison keywords, see “SQL Keywords” in the “SQL Reference” chapter.

Extending a Comparison

SQL supports the boolean operators AND, OR, and NOT. In the previous examples, you have seen how the NOT operator can be used to form a negative comparison. You can use the AND and OR operators in comparisons to extend the conditions required by the WHERE clause.

The boolean operators are simple true/false operations:

AND	Both the preceding and the next comparisons must be true
OR	Either the preceding or the next comparison must be true
NOT	The comparison must be false

For example,

```
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE > 30.00 AND
DESCRIPTION LIKE "ROYAL%"
```

will return a results table of a DESCRIPTION column for each row having both a PRICE greater than \$30.00 and a DESCRIPTION beginning with "ROYAL". If either comparison does not find a value, the entire WHERE clause is false (does not return a row).

On the other hand, the OR operator requires that only one of the comparisons find a value. For example, the query

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
WHERE DESCRIPTION LIKE "ROYAL%" OR
DESCRIPTION LIKE "RED%"
```

returns a results table of a DESCRIPTION and PRICE column for each row having a DESCRIPTION of value beginning with "ROYAL" or "RED". Only one of the comparisons needs to find a match in order for the WHERE clause to return a value.

Ordering Selected Data

The rows selected by your queries will not be in any particular order. To return the rows sorted as you would like, use the ORDER BY keyword.

For example, the following query returns a results table of all rows in the PRODUCTS table, with the rows sorted from lowest to highest price:

```
SELECT * FROM PRODUCTS
ORDER BY PRICE
```

By default all rows are sorted from lowest to highest values. There are two keywords you can use to specify the sort order:

- ASC (the default)
- DESC

To reverse the ordering of the previous example, use the statement:

```
SELECT * FROM PRODUCTS
ORDER BY PRICE DESC
```


The results table returned would look like this:

```
104.00
35.90
30.00
25.95
25.95
```

You can specify more than one column to use in sorting the selections. Subsequent columns are ordered within the preceding sort. For example, in the query

```
SELECT * FROM PRODUCTS
ORDER BY PRICE, DESCRIPTION
```

all of the rows are sorted according to the PRICE value. In addition, where more than one product has the same price, these rows are also sorted based on the DESCRIPTION value. The results table would look like this:

010100	RED SAILBOAT TIE	25.95
010120	TEAL PAISLEY TIE	25.95
010210	BLACK LEATHER GLOVES, MEN'S	30.00
020120	ROYAL SILK SCARF	35.90
030460	MULTI SILK SWEATER	104.00

You can also apply ordering criteria to conditional selections (the rows returned by a WHERE clause), as in:

```
SELECT * FROM PRODUCTS
WHERE PRICE < 32.50
ORDER BY DESCRIPTION
```

The results table returned consists of all rows with a PRICE value less than \$32.50. The rows are sorted based on the DESCRIPTION value.

☞ For more information about ORDER BY, see “SQL Keywords” in the “SQL Reference” chapter.

Grouping Selected Data

You may want your query to involve only selected groups of data. To define groups, use the GROUP BY keyword. GROUP BY collects all of the selected rows into groups based on a particular column or columns.

For example, the query

```
SELECT PRICE, COUNT(*) FROM PRODUCTS
GROUP BY PRICE
```

will group all products by price and return the number of rows having identical PRICE values for each group. The results table might look like this:

25.95	2
30.00	1
35.90	1
104.00	1

To order the selected data, follow the GROUP BY clause with an ORDER BY clause:

```
SELECT PRICE, COUNT(*) FROM PRODUCTS
GROUP BY PRICE
ORDER BY PRICE DESC
```

The results table of handle types is grouped by cost. In addition, each group is sorted in descending order:

104.00	1
35.90	1
30.00	1
25.95	2

Setting Conditions for Groups

If you want the selected groups to meet certain criteria for the group as a whole, you can follow the GROUP BY clause with a HAVING clause.

The HAVING keyword is much like the WHERE keyword: the syntax for their clauses are exactly the same. The only difference is that HAVING is used for testing groups, and WHERE for individual rows.

For example, the query

```
SELECT PRICE, COUNT(PRICE) FROM PRODUCTS
GROUP BY PRICE
HAVING PRICE > 30.00
```

will group the rows of PRICE values. The groups are then tested for a PRICE value greater than \$30.00. The query returns a results table of PRICE columns consisting of each group meeting the range test.

The results table would look like this:

35.90	1
104.00	1

Forming Unions

You can combine tables to produce a results table containing all of the rows selected from each table. To combine the tables, use the UNION keyword.

For example, suppose the PRODUCTS table for the previous year was PRODUCTS_88, and the column definitions had not changed:

```
SELECT * FROM PRODUCTS_88
UNION
SELECT * FROM PRODUCTS
```

This union of the two tables will select all rows from both tables. Any duplicate rows will not appear in the results table.

When UNION is followed by the ALL keyword, all rows selected by the queries, including any duplicate rows, will appear in the results table. The previous example would contain any duplicate rows rewritten as:

```
SELECT * FROM PRODUCTS_88
UNION ALL
SELECT * FROM PRODUCTS
```

You can use any of the appropriate keywords in your unions, except ORDER BY:

```
SELECT DESCRIPTION FROM PRODUCTS_88
WHERE PRICE < 30.00
UNION
SELECT DESCRIPTION FROM PRODUCTS
WHERE PRICE < 30.00
```

The results table produced by this union will allow you to compare differences in the cost of products.

Performing Multiple Unions

To combine the results of more than one union, you can perform a union operation on a subsequent union. Use parentheses to associate the unions. For example, assume a PRODUCTS table for 1987 as well:

```
SELECT * FROM PRODUCTS_88
UNION
(SELECT * FROM PRODUCTS_87
UNION
SELECT * FROM PRODUCTS)
```

The union will produce a results table consisting of all rows from all three tables.

Joining Tables

When you need to select particular information from more than one table, you can use a single query. This is called *joining* the tables.

In order to be able to usefully join tables, you need to be able to perform meaningful comparisons between at least one column in each table. Columns having identical values, such as foreign keys, are often used to join tables. This “join” column gives you a basis for finding relationships between the data in the tables.

Suppose that in addition to the PRODUCTS table, there is another table in the database called INVOICES. The table might look like this:

INVOICES

INVOICE_NO	CUSTOMER_ID	CODE	QUANTITY
1000	330	020120	1
		030100	1
1001	349	040130	2
		010120	1
		050140	1
		040100	1
1002	298	030300	1
		030360	2
1003	134	060300	1
		030220	1
1004	125	050100	1
		050320	2

In this case, the join column in the INVOICES table is a foreign key, since the CODE column values correspond to those for the primary key column PRODUCT_CODE of the PRODUCTS table.

To join the tables, you might want to use the full name for the columns in your query to ensure there is no ambiguity. You will also need to list the tables referenced.

For example:

```
SELECT PRODUCTS.DESCRPTION, PRODUCTS.PRICE,
INVOICES.QUANTITY FROM PRODUCTS, INVOICES
WHERE INVOICES.CODE = PRODUCTS.PRODUCT_CODE
```

The results table for the query will consist of the DESCRIPTION and PRICE columns from the PRODUCTS table and the QUANTITY column from the INVOICES table. Only the rows having matching CODE and PRODUCT_CODE columns are selected.

Using Range Variables

Sometimes, for reasons of brevity or ease of reference, it is helpful to be able to assign a different, temporary name to a table. These are called *range variables*, and are valid only for the duration of the query.

Range variables are assigned by following the table name in a query with the range variable. Used in the previous example, the range variables might be:

```
SELECT PRODUCTS.DESCRPTION, PRODUCTS.PRICE,
INVOICES.QUANTITY FROM PRODUCTS T1, INVOICES
T2
WHERE T1.PRODUCT_CODE = T2.CODE
```

Nesting Queries

The WHERE clause in your queries can use the results table returned by another query for its data. These nested queries (*subqueries*) enable you to base the conditions for your query on data selected from other tables or views.

You may recall that the format for a WHERE clause is:

```
WHERE expression comparison expression
                        subquery
```

As noted previously, the *expressions* might be literal values, columns or column equations.

- ✦ As opposed to a main query, which can evaluate multiple *expressions*, a subquery can evaluate only a single *expression*, unless you use the ALL, ANY, or SOME comparison keywords (see “Using Comparison Keywords with Subqueries” below). This also restricts your use of the * (asterisk) operator to existence queries (see “Making Existence Queries” below).

Subqueries are always enclosed in parentheses. You can apply comparison operations to your subqueries, much as you would to a column or literal value.

For example, the WHERE clause in the query

```
SELECT PRODUCT_CODE, DESCRIPTION FROM PRODUCTS
WHERE PRODUCT_CODE = ANY
  (SELECT DISTINCT CODE FROM INVOICES)
```

compares the product codes in the PRODUCTS table with the codes for the products ordered. The results table returned is a list of the codes and products that have been ordered. Duplicate orders for a particular product code are not included.

Using Comparison Keywords with Subqueries

There are a few comparison keywords you can use only with subqueries. These are:

- ALL
- ANY/SOME

These keywords are always preceded by a comparison operator and followed by a subquery.

All Values

The ALL keyword indicates that the operation must be successful against all values returned by the subquery.

For example,

```
SELECT DESCRIPTION, PRICE, QUANTITY FROM
PRODUCTS, INVOICES
WHERE INVOICES.QUANTITY > ALL
  (SELECT QUANTITY FROM INVOICES_88)
```

Every product that has sold more this year than any single product last year is selected.

Some Values

The keywords **ANY** or **SOME** are synonyms, identical in their usage and effect. They specify that the evaluation need only be successful against any of the values returned by the subquery.

Both keywords will return each value within range of the comparison operator.

Used with **ANY**, the previous example,

```
SELECT DESCRIPTION, PRICE, QUANTITY FROM
PRODUCTS, INVOICES
  WHERE INVOICES.QUANTITY > ANY
    (SELECT QUANTITY FROM INVOICES_88)
```

Every product that has sold more this year than at least one of the products sold last year is selected.

Making Existence Queries

SQL gives you an easy way to find out if a row exists in a table. The **EXISTS** keyword tests the result of a subquery. If the subquery returns a non-null value (a result table that is not empty), **EXISTS** returns true. If the subquery does not find the row, **EXISTS** returns false.

Since a row is tested only for its existence and not for a value, the existence query usually uses the ***** (asterisk) operator.

For example, the query

```
SELECT PRODUCT_CODE, DESCRIPTION, PRICE FROM
PRODUCTS
  WHERE NOT EXISTS
    (SELECT * FROM PRODUCTS_88
     WHERE PRODUCTS.PRODUCT_CODE =
       PRODUCTS_88.PRODUCT_CODE)
```

tests for the existence of each product *not* sold last year, before selecting the code, description and price for the new product.

Selecting Values from Multivalued Columns

The **KEYQUAL** column, automatically defined for multivalued columns, uniquely identifies each value in a multivalued column. Using the **KEYQUAL** column to define the search conditions for a **WHERE** clause enables you to select any values in the column.

For example, you could select a specific value with the query:

```
SELECT CODE, QUANTITY FROM INVOICES
WHERE CUSTOMER_ID = 349 AND KEYQUAL = 2
```

Or you might select all of the values in the multivalued columns:

```
SELECT CODE, QUANTITY FROM INVOICES
WHERE CUSTOMER_ID = 349
```

In this case, you do not need to reference the **KEYQUAL** column, since all of the **CODE** and **QUANTITY** values have the **CUSTOMER_ID** column in common.



Chapter K5

Modifying the Database

This chapter describes how to add, change, or remove information from your database.

Inserting Rows	K 5 . 1
Updating Rows	K 5 . 4
Deleting Rows	K 5 . 6
Deleting Tables and Views	K 5 . 8



Inserting Rows

Having created your tables and views, the structure of your database, you need a way to enter new rows of data into the tables. To do this, use the INSERT command.

When you insert a row, it is important that you know the columns defined for the table. The rows that you are insert are simply an association of values to be assigned to these columns. This means that the values you assign must be of the required data type for the columns.

There are a few ways you can assign values to a column when you insert a row into a table:

- Specify the columns and value assignments
- List the values to be assigned to the columns without specifying the columns
- Use the results of a query for the column assignments

Assigning Values to Specified Columns

The INSERT command is always followed by the keyword INTO and the name of the table. The basic format of an INSERT statement is:

```
INSERT INTO table columns VALUES values
```

The PRODUCTS table used in previous examples might look something like this without any rows:

PRODUCTS

PRODUCT_CODE	DESCRIPTION	PRICE
--------------	-------------	-------

To insert a new row into the PRODUCTS table, you could enter:

```
INSERT INTO PRODUCTS (PRODUCT_CODE, DESCRIPTION, PRICE)
VALUES (010101, "GREEN SAILBOAT TIE", 25.95)
```

The PRODUCTS table would then appear like this:

PRODUCTS

PRODUCT_CODE	DESCRIPTION	PRICE
010101	GREEN SAILBOAT TIE	25.95

Assigning Values to Unspecified Columns

You can achieve the same effect as above by using a slightly different syntax for your INSERT statement. The format for an INSERT statement may be expanded to:

```
INSERT INTO table VALUES values
```

The VALUES keyword introduces the values you want to assign to the columns (a VALUES 'clause'). The order in which you list the values determines their column assignment. The order must correspond to the order of the desired columns in the table.

The previous example can be rewritten as:

```
INSERT INTO PRODUCTS VALUES (010101, "GREEN SAILBOAT TIE",  
25.95)
```

Inserting Rows with Multivalued Columns

When you insert rows having multivalued columns, each value in a multivalued column must have a unique KEYQUAL value. If you do not specify a KEYQUAL value, sequential values will be automatically assigned.

- ✦ The KEYQUAL column is automatically defined when you create a table having a multivalued column. For more information, see "Allowing Multiple Values" in the chapter "Building a Database".

To specify a KEYQUAL value when inserting a row, use a WHERE clause:

```
INSERT INTO INVOICES (CUSTOMER_ID, CODE, QUANTITY, KEYQUAL)  
VALUES (100, 010101, 1, 1)
```

This example inserts a row with a three column assignments. A quantity of 1 is assigned as the first value in the multivalued QUANTITY column.

Without specifying a KEYQUAL value, the previous example would be:

```
INSERT INTO INVOICES (CUSTOMER_ID, CODE, QUANTITY) VALUES  
(100, 010101, 1)
```

The first value in the multivalued QUANTITY column is given a value of 1, and an identifying KEYQUAL column value is also assigned a value of 1.

Adding Values to Multivalued Columns

Because relationships to the non-multivalued column values are implied when you add values to existing multivalued columns in the row, you need specify only the values you wish to add.

When you first insert a row into the INVOICES table, your statement might be as in the previous example:

```
INSERT INTO INVOICES (CUSTOMER_ID, CODE,
  QUANTITY) VALUES (100, 010101, 1)
```

Your subsequent INSERT statements do not need to specify assignments to the CUSTOMER_ID column: this relationship is implied for all values in the multivalued CODE and QUANTITY columns. The INSERT statement might be:

```
INSERT INTO INVOICES (CODE, QUANTITY) VALUES
(010100, 2)
  WHERE CUSTOMER_ID = 100
```

- ✎ It is not an error to specify the non-multivalued columns for a particular row in your INSERT statements. However, if there is any variation in these column references, Advanced Revelation SQL will not allow the INSERT operation.

Notice as well, that a KEYQUAL assignment is not required. You can instead allow these assignments to be made automatically. The KEYQUAL assignment is determined by incrementing the preceding KEYQUAL value by 1.

Inserting Rows Using a Query

You can use the results table returned by a query to provide the data to insert. The INSERT operation will assign the column values in the results table to any columns you specify.

- ✎ The query may not reference the same table as the INSERT statement.

The format for an INSERT statement can be further expanded to:

```
INSERT INTO table columns query
```

Suppose that you had another table named NEW_PRODUCTS. You might then use the PRODUCTS table as a source of data for the new table:

```
INSERT INTO NEW_PRODUCTS (PRODUCT_CODE, ITEM)
SELECT PRODUCT_CODE, DESCRIPTION FROM PRODUCTS
```

This statement assigns the values of the PRODUCT_CODE and DESCRIPTION columns in the PRODUCTS table to the specified columns in the NEW_PRODUCTS table. Notice that the column names may or may not be the same.

- ✎ The column data types and domains however, must be the same.

Using INSERT to Copy a Table

To copy all of the rows from one table to another 'compatible' table (having corresponding columns and data type definitions), use an * (asterisk) operator in your query.

This completes all of the format options for an INSERT statement:

```
INSERT INTO table query
```

The previous example could now be rewritten as:

```
INSERT INTO NEW PRODUCTS  
SELECT * FROM PRODUCTS
```

The column names and definitions are inserted exactly as they exist in the table referenced by the query.

- ☞ For more information see INSERT in the "SQL Statements" part of the "SQL Reference" chapter.

Updating Rows

Most databases change, new data is constantly being added. In order to maintain current data in your tables, use the UPDATE command.

- ☞ You cannot update a column that is the primary key. Any attempt to do so will cause an error message to display.

UPDATE changes the column values of your existing rows. You specify the name of the table to update, the SET keyword, and the column assignments. An UPDATE statement can assign a new column value:

- To a column in every row
- Conditionally, to columns in specific rows
- To multiple columns
- To multivalued columns

The UPDATE command gives you a number of options to use in specifying the values to assign to a column. A value assignment can be:

- A literal value
- The name of a column
- A column equation
- The NULL keyword

Updating a Column in All Rows

The basic format of an UPDATE statement is:

```
UPDATE table SET column = value
```

Using this format, you can make changes to a column in every row of a table. For example, to assign a new value to the **QUANTITY** column in all rows of the **INVOICES** table:

```
UPDATE INVOICES SET QUANTITY = 0
```

Statements of this type allow you to quickly make global updates to a table.

Updating with Conditions

Often you will want your UPDATE statements to apply only to certain rows. Using a **WHERE** clause, you can specify the conditions for updating a column. The format for a conditional update is:

```
UPDATE table SET column = value WHERE conditions
```

For example,

```
UPDATE INVOICES SET QUANTITY = (QUANTITY + 1)
WHERE CODE = 010100
```

Only the rows having a **CODE** value of 010100 will be updated to the new **QUANTITY** value.

Updating Multiple Columns

The UPDATE command does not restrict you to updating a single column at a time. You can designate multiple column assignments by delimiting each assignment clause (*column = value*) with a comma. For example,

```
UPDATE CUSTOMERS SET ADDRESS = 831 116TH AVE, PHONE =
"415-223-9077"
WHERE CUSTOMER_ID = 330
```

will update the **ADDRESS** and **PHONE** columns for the customer having a **CUSTOMER_ID** value of 330.

Updating Multivalued Columns

The **KEYQUAL** column, automatically defined for multivalued columns, uniquely identifies each value in a multivalued column. Using the **KEYQUAL** column to define the search conditions for a **WHERE** clause enables you to update any values in the column.

For example, the statement,

```
UPDATE INVOICES SET QUANTITY = 3
  WHERE CUSTOMER_ID = 193 AND KEYQUAL = 2
```

will update only the **QUANTITY** value identified by the **KEYQUAL** value of 2.

To update more than one value in the multivalued column, you could use the statement:

```
UPDATE INVOICES SET QUANTITY = 2
  WHERE CUSTOMER_ID = 349 AND KEYQUAL > 1
```

- ✎ You cannot specify multiple values to use in updating the values in a multivalued column.

You can update all values in a multivalued column by simply not specifying **KEYQUAL**:

```
UPDATE INVOICES SET QUANTITY = 3
  WHERE CUSTOMER_ID = 349
```

Deleting Rows

As your database changes, you may need to remove rows from your tables. Using the **DELETE** command, you can remove from a table only those rows meeting specified conditions, or all of the rows.

In order to remove the proper row from a table, the row must have a column value that distinguishes it from all other rows in the table. In Advanced Revelation SQL each row must have a unique, primary key. This feature guarantees that you are able to properly reference a row.

The **DELETE** command is always followed by the keyword **FROM** and the name of the table containing the row(s) to delete.

Deleting Rows upon Condition

To establish the conditions for removing rows from a table, use a WHERE clause. The format for a conditional DELETE statement is:

```
DELETE FROM table WHERE conditions
```

For example,

```
DELETE FROM INVOICES  
WHERE CUSTOMER_ID = 336
```

Deleting Rows with Multivalued Columns

If a row contains multivalued columns, deleting the row will delete only the first value in multivalued columns. To delete all values, you will need to treat each value in a multivalued column as though it were in a new row. Each of these fictional rows however, uses the same primary key.

- The number of rows corresponds to the largest number of values in any single multivalued column for the row.

In other words, you must delete the row identified by a specific primary key once for each value in the largest multivalued column.

For example, to delete an INVOICES row for a customer that has ordered two items, you would repeat the following DELETE statement twice:

```
DELETE FROM INVOICES  
WHERE CUSTOMER_ID = 298
```

Deleting All Rows

To remove all of the data from a table, use the format:

```
DELETE FROM table
```

For example,

```
DELETE FROM INVOICES
```

removes all rows in the INVOICES table.

The existing column definitions are not affected when you delete the rows, so you can reuse the table (insert new rows) if you wish.

Deleting Tables and Views

When the tables or views you have created are no longer useful, you can remove them.

To remove a table or view, use the **DROP TABLE** or **DROP VIEW** commands. The command is followed by the name of the table to delete.

☞ Deleting a base table does not delete any related views.

For example,

```
DROP TABLE INVOICES
```

will delete the INVOICES table.

To remove the view **CUSTOMER_INVOICES**, that uses both the INVOICES and CUSTOMERS tables in its definition, enter the statement:

```
DROP VIEW CUSTOMER_INVOICES
```

Chapter K6

Managing the System

This chapter is intended primarily for the advanced user or system administrator. Here you will find information for establishing, controlling, and maintaining your SQL environment.

The SQL Environment K 6 . 1

Accessing the Dictionary K 6 . 7



The SQL Environment

The importance in SQL of transaction processing and concurrency controls requires adequate parameters or settings to adjust these processes. Advanced Revelation gives you the flexibility to configure your SQL environment according to your needs.

The discussion here applies only to SQL. Although some of the environmental settings are used outside of the SQL environment as well, the available functionality may be different. You can find these settings in the Global Environment window and the Concurrency Control Environment window.

Establishing the Global Environment

There are some environmental settings that have global effect. These involve transaction processing and locking issues.

- ☛ If you are using a network it is very important to coordinate these settings on all workstations.

The global settings are available at the Global Environment window. To display the Global Environment window, choose the **Management-Environment-Global** sequence of options from the Main menu.

Assigning a Commit Log Volume

The commit log is the central record of all transactions currently being committed. To assign a volume, use the *Commit Log Volume* prompt. The default setting is the TRANSACT volume.

- ☛ All stations on a network must use the same commit log volume.

Queing Committed Transactions

You can protect data integrity on the network by queuing transactions as they are committed. The transactions are assigned an entry in a file queue and are only written to the target file after any previous entries.

This ensures data integrity in two ways:

- The data cannot be changed for the duration of the transaction.
- The transaction is protected against system failures happening after the commit, but before it is selected from the file queue.

To control queuing use the *Commit Protection* prompt. The default setting is 'Yes'.

If *Commit Protection* is set, you can guarantee serializable transactions by also setting the *Consistency Level* prompt at the Concurrency Control Environment window to 4.

- ☞ This is the only method available to ensure serializable transactions in Advanced Revelation. For more information, see "Setting a Consistency Level" below.

You can display queuing information using the TCL command COMMITLOG. From the SYSPROG account, use this display to remove an entry from a file queue if necessary.

- ☞ For more information about the commit log, see the "Managing Files and Records" chapter of *Filing Systems*.

Resolving Deadlocks

When a deadlock situation is detected, one of the conflicting transactions must be rolled back. To determine which transaction is selected, use the *Deadlock Resolution Scheme* prompt.

- ☞ For information on deadlocks, see "Detecting Deadlocks" below.

You may choose to have transactions selected based on:

- The Youngest or most recent transaction
- The Least Active transaction, as determined by the Activity Index: an index of I/O activity. You can display the activity index using the TCL command COMMITLOG.

The default setting for the *Deadlock Resolution Scheme* prompt is 'Least Active'.

- ☞ For more information, see the "Locking" chapter in *Networks*.

Enabling Implicit Locking

Unless you disable locking, by using a *Consistency Level* setting of 0, implicit (automatic) locking is active for SQL transactions.

- ☞ For more information, see "Setting a Consistency Level" below.

You can control the type of locking used by setting the *Implicit Locking* prompt to 'Record' or 'File' locking. The default setting is record locking.

Establishing Concurrency Controls

Concurrency controls manage the issues of conflicting or concurrent access to data. These controls have application primarily for networks.

To display the Concurrency Control Environment window, choose the **Management-Environment-Concurrency** sequence of options from the Main menu.

Assigning a Transaction File Volume

The temporary files used for a workstation's transactions must exist in a single volume. To assign a volume, use the *Transaction File Volume* prompt. The default setting is the TRANSACT volume.

Releasing All Locks at Commit

You can choose to release *all* locks when the workstation commits a transaction. This is helpful in terms of performance, since the system does not have to perform any of the normal checking for lock release.

However, this setting should be used carefully. Every lock will be release, whether implicitly or explicitly set. Due to the interactive nature of Advanced Revelation, the system must often set locks when you move between environments (from the Editor to TCL, for example). Releasing a lock in this case may allow data to be corrupted. To avoid the possibility of releasing these locks, do not allow open records that may be affected. If you wish to guarantee maximum control over unlocking, do not use this setting.

To enable release of locks at commit, use the *Unlock All at Commit* prompt. The default setting is 'No'.

Beginning Transaction Processing Automatically

A workstation needs to specify the beginning and end of a transaction. The end of a transaction is signalled when the user executes a COMMIT statement. There are two methods that you can use to indicate the beginning:

- Use the BEGIN TRANSACTION statement.
- Enable automatic transaction processing for the workstation. A transaction is then initiated when the user accesses a table.

To begin transactions automatically, use the *Auto-Transaction Start* prompt. The default setting is 'No'.

You can use the TRANSACTION command at TCL to override this setting. However, when you reset the system or log in again, the *Auto-Transaction Start* setting is used.

☞ For more information, see "Transaction Processing" in the "Managing Files and Records" chapter of *Filing Systems*.

Detecting Deadlocks

A deadlock situation occurs when two or more stations on the network each attempt to lock data that the other already has locked. Deadlock detection is done automatically in Advanced Revelation SQL.

Posting a Deadlock

When a deadlock is detected, you can determine the amount of time before the workstation's locks are posted to the central deadlocks file (@DEADLOCK). This listing of locks is used by the deadlock checking process (see "Checking Deadlock Status" below).

To control the delay before posting a deadlock, use the *Deadlock Posting* prompt. The default value is 15 seconds.

Checking Deadlock Status

The deadlock checking process is triggered when the workstation posts a deadlock occurrence. In order to determine whether a deadlock is still in effect for the workstation, the system must continually check the postings in the central deadlocks file.

You can specify the time interval the deadlock checking process will use between each check. To do so, use the *Deadlock Checking* prompt. The default interval is 10 seconds.

Controlling the Lock Wait Process

When the workstation is refused a request for a lock, it will continue indefinitely to attempt the lock, unless otherwise instructed. This is termed a 'lock wait' process.

At the end of the lock wait process, the current statement is aborted. The user then has the option of committing, rolling back, or ignoring the transaction. Ignoring the transaction aborts the current statement, but does not affect the transaction.

You have a few ways to handle the lock wait process:

- Specify an interval to use between attempts to place the lock.
- Allow the user to manually end the process.
- Disable the user's ability to manually end the process.
- Disable the process.

Specifying an Interval

To determine the time interval for the workstation to use between attempts to place a lock, set the *Lock Timeout* prompt. The default interval is 50 seconds.

Manually Ending the Lock Wait Process

At any time during the lock wait process, the user can end the process by pressing [Esc], unless you have disabled this (see below).

Disabling the Lock Interrupt

You can disable the user's ability to interrupt the lock wait process by setting *Disable Lock Interrupt*. The default setting is 'No'.

Disabling the Lock Wait Process

To disable the lock wait process entirely, set the *Lock Timeout* prompt to zero.

Setting a Lock Limit

You can limit the maximum number of locks held by the workstation by setting the *Lock Limit* prompt.

The default setting of zero disables a lock limit. The workstation may then use the maximum number of locks available to the network. When the limit for a station is exceeded, the next lock fails.

☞ The number of locks you may use is dependant upon your network. Refer to your network documentation.

Setting a Consistency Level

A 'consistency level' determines both the level of locking used by the workstation and the duration of the lock (when locks are released).

To assign a default consistency level, use the *Consistency Level* prompt. The SET TRANSACTION statement may be used to change the default value for the duration of the logon session.

The following are the consistency level settings available:

- Level 0* A setting of zero completely disables any implicit locking. In order to lock records and tables, you must lock them explicitly using R/BASIC commands. You should not choose this consistency level unless you are working with a single-user system.
- Level 1* A setting of 1 causes SQL to set exclusive locks during updates of tables or records. The lock is held for the duration of the update operation. No locking takes place during queries (reads).

Operation	Lock Level	Duration
Update	Exclusive	Update
Read	none	none

Level 2 A setting of 2 sets exclusive locking during updates for the duration of each transaction. When the transaction is committed or rolled back, the locks are released. No locking is done for queries (reads).

Operation	Lock Level	Duration
Update	Exclusive	Transaction
Read	none	none

Level 3 A setting of 3 sets exclusive locks during updates for the duration of each transaction. In addition, it sets shared locks during queries (reads) for the duration of each query.

Operation	Lock Level	Duration
Update	Exclusive	Transaction
Read	Shared	Query

Level 4 A setting of 4 sets exclusive table locks during updates for the duration of each transaction, and shared table locks during queries for the duration of the transaction.

Operation	Lock Level	Duration
Update	Exclusive	Transaction
Read	Shared	Transaction

Because this setting can potentially use a large number of locks, the *Implicit Locking* prompt at the Global Environment window must be set to 'File Locking' to use this level.

Serializable transactions are guaranteed with level 4 when *Commit Protection* is set at the Global Environment window. This is the only method available to ensure serializable transactions in Advanced Revelation.

For more information, see the "Locking" chapter in *Networks*.

Accessing the Dictionary

Advanced Revelation SQL maintains a master dictionary: a table of the dictionary definitions for all currently attached tables. The dictionary is stored in the SYSCOLUMNS table.

To display all of the definitions in the SYSCOLUMNS table, use the TCL command LISTDICT. At TCL, enter:

```
LISTDICT SYSCOLUMNS
```

Use the SQL window to find definitions for particular tables.

Finding Table Definitions

To select the dictionary for a specific table, query the SYSCOLUMNS table for the table information. For example,

```
SELECT * FROM SYSCOLUMNS  
WHERE TABLE_NAME = "CUSTOMERS"
```

will return the entire dictionary for the CUSTOMERS table. To access selected definitions or attributes for a table, base your query on one of the columns in SYSCOLUMNS. For example,

```
SELECT COLUMN_NAME FROM SYSCOLUMNS  
WHERE TABLE_NAME = "CUSTOMERS "
```

returns all of the column names for the CUSTOMERS table. You can also find definitions for multiple tables. The query,

```
SELECT COLUMN_NAME FROM SYSCOLUMNS  
WHERE REPEATING = "M"
```

returns the names of all multivalued columns in all tables.

- ☛ For more information about dictionaries, see “Advanced Revelation Dictionaries” in *Filing Systems*.



Chapter K7

Programming with SQL

This chapter is a guide to the use of embedded SQL with the R/BASIC procedural language. The SQL Module Language is also described.

Introduction	K 7 . 1
The SQL Module Language	K 7 . 1
Embedded SQL	K 7 . 2



Introduction

In addition to using SQL as an interactive language, you can use a compiled version of SQL. Compiled SQL is a language very similar to interactive SQL, but one that offers several benefits to you as an application developer:

- Faster execution of SQL statements, since these will be pre-compiled.
- The ability to pass variable information to an SQL statement. This enables you to establish a program that can prompt a user for input, and execute an SQL statement using the input your program receives.
- Procedural control over SQL statements, allowing you to control precisely when and how SQL statements are executed.
- The ability to test for various error conditions that can occur during the execution (or attempted execution) of an SQL statement.
- The ability to use R/BASIC operations (such as dynamic array operations) to manipulate the results of SQL statements.
- The general advantage of SQL functionality from within R/BASIC. This enables you to execute powerful high-level query and update statements without having to code them yourself.

Compiled SQL is available in a form frequently referred to as “SQL Module Language.” SQL Module Language is a standalone language that you can compile using the standard R/BASIC compiler.

As an alternative, you can write, or “embed,” SQL statements into your R/BASIC programs. This is a very convenient and powerful way for you to program using SQL. In fact, it is the most common way to program with SQL.

The SQL Module Language

Advanced Revelation permits you to create complete SQL modules as defined in the ANSI standard for compiled SQL. You can create, compile, and catalog these modules in much the way you work with R/BASIC programs. Once a module is complete, you can call it in the same manner that you call an external R/BASIC subroutine.

One particular advantage to using SQL modules is that they are portable. Because the SQL Module Language follows a defined industry standard, you can transfer your modules from one SQL-compatible platform to another.

Although the SQL Module Language is a complete language for describing SQL in compile form, it is not a procedural language. There is no provision for branching or looping, testing, or other fundamental programming constructs.

Instead, SQL modules are meant to be called as external routines from another programming language. Procedural control is the responsibility of this second language, with the execution of SQL statements the responsibility of the SQL module itself.

Embedded SQL

One way to use compiled SQL is to create SQL modules and call them as external routines from another language. A far easier way to do this is to use “embedded” SQL.

In general terms, embedded SQL refers to the ability to embed SQL statements directly into a “host language.” In the various platforms on which SQL is available, this host programming language can be FORTRAN, COBOL, PL/1, Pascal, or C. In Advanced Revelation, the host language is R/BASIC.

To use embedded SQL, you can write certain SQL statements directly into your R/BASIC code, interleaving them with standard R/BASIC statements and functions. The SQL statements you embed in your R/BASIC programs use slightly different syntax than their counterparts in interactive SQL. There are also a few statements that are unique to embedded SQL.

☞ The SQL Reference always notes if a statement is unique to embedded SQL.

When you compile an R/BASIC program that contains embedded SQL statements, the compiler automatically creates the appropriate SQL module for you. At the same time, it substitutes call statements to this module in your original program. The end result is the same as if you had coded the SQL module yourself, and coded calls to into your program. However, everything happens automatically, and is therefore much easier and quicker.

You can use embedded SQL statements at any point in any R/BASIC program. As noted above, there are special forms of some SQL statements for use with embedded SQL. Details on these statements appear below and in the “SQL Reference” chapter.

Types of Statements

There are 14 statements available in embedded SQL. In most cases, these are statements that are similar to statements that you can execute interactively. However, a few are unique to embedded SQL.

Embedded SQL makes a general distinction between two different classes of statements. The first class is simply the embedded SQL version of interactive statements. This set of statements includes UPDATE, INSERT, DELETE, and a special version of SELECT. There are slight differences between these statements and their interactive counterparts.

A second class of embedded SQL statements creates and maintains a logical table called a “cursor.” Using cursor statements, you can fetch rows into your program, or perform updates or inserts of tables.

☞ The use of cursors with embedded SQL is discussed in more detail later in this chapter.

The full set of statements that you can use with embedded SQL is summarized in the table below. More detail about each statement is found in the “SQL Reference” chapter. In addition, details about using some of the statements — for example, details about cursors — appear later on in this section.

Table K7-1

Embedded SQL Statements

Statement	Operation
SELECT	Select one or more rows from a table into a variable or variables
UPDATE	Update a row or rows in a table
INSERT	Insert a row or rows in a table
DELETE	Delete a row or rows from a table
DECLARE	Initialize a cursor with a cursor variable and a select condition
OPEN	Load a cursor by executing the query expression indicated in a DECLARE statement
FETCH	Extract a row out of a cursor
UPDATE ... CURRENT	Update a row using a cursor
DELETE ... CURRENT	Delete a row using a cursor
CLOSE	Release a cursor
BEGIN TRANSACTION	Begin a transaction
COMMIT WORK	Commit an open transaction
ROLLBACK WORK	Roll back a pending transaction and cancel changes
WHENEVER	Establish default behavior for error conditions

Entering Statements

All embedded SQL statements must be prefaced with the keywords “EXEC SQL.” The SQL statement then immediately follows, without quotes. One SQL statement follows each EXEC.

You must explicitly end the embedded SQL statement with a semicolon (;). If the SQL statement crosses multiple lines, use the keywords “END EXEC” to end the statement. You may not put blank lines between portions of an embedded SQL statement.

Here is a small program fragment that illustrates how embedded SQL statements appear in an R/BASIC program:

```
EXEC SQL DECLARE CURSOR TEST_CURSOR FOR
  SELECT * FROM CUSTOMER
END EXEC
EXEC SQL OPEN TEST_CURSOR;
RECORD = ""
LOOP UNTIL SQLSTATE = 100
  EXEC SQL FETCH TEST_CURSOR INTO :RECORD;
  CALL MSG("RECORD = ":RECORD, "", "", "")
REPEAT
```

Note that each embedded SQL statement is preceded with the EXEC SQL keywords, and ends with a semicolon. One statement (DECLARE) crosses line boundaries, and therefore ends with "END EXEC."

Using Variables

One of the advantages of using embedded SQL is that you can pass information to and from the statements using variables. You can use variables to pass dynamic information to an embedded SQL statement, such as literal values for a query. You can also have SQL statements return data into local variables so that you can manipulate it further.

- ☛ You cannot use @variables (for example, @RECORD) in embedded SQL statements.

All variables in an embedded SQL statement must be preceded with a colon. This distinguishes them from keywords and column and table names. For example, the following program fragment illustrates the use of variables in an embedded SELECT statement:

```
NEW_VALUE = ""
CALL MSG("Enter new value for CITY", "R", NEW_VALUE, "")
IF NEW_VALUE THEN
  EXEC SQL UPDATE CUSTOMER SET CITY = :NEW_VALUE;
END
```

Passing Data to a Statement

You can substitute a variable in an embedded SQL statement for any "scalar expression." That is, you can use a variable anyplace that you would normally insert a literal string or numeric value in an interactive SQL statement. Variables can therefore appear at these positions in embedded SQL statements:

- The literal values in a WHERE clause
- The literal values in a HAVING clause
- A literal value in a SET clause
- A literal value in a VALUES clause

Returning Data from a Statement

In addition to using variables to pass data to a statement, you can have the statement return values to you in a variable. This capability is used to return one or more rows to your program out of a table or a cursor.

Variables are used in the embedded SELECT and FETCH statements. In both cases, you specify a "target parameter" (R/BASIC variable) where the statement is to return the row(s):

```
ST = "" ; RECORD = ""
CALL MSG("Enter the state", "R", ST, "")
IF ST THEN
    EXEC SQL SELECT * INTO :RECORD
        FROM CUST WHERE ST = :ST
    END EXEC
END
```

In this example, any row(s) returned by the SELECT statement are placed into the variable RECORD. If there are multiple rows, they are delimited with a record mark (@RM).

ANSI-standard SQL calls for there to be a variable corresponding to each column returned by a SELECT or FETCH statement. For example, if you anticipate returning the columns CITY and ST from your SELECT, you specify two variables after INTO, much like this:

```
CITY = "" ; ST = ""
EXEC SQL SELECT CITY, ST INTO :CITY, :ST
    FROM CUSTOMER WHERE ST = "WA"
END EXEC
```

In R/BASIC, however, you do not need to specify multiple variables. The columns will all be returned into the target parameter as a dynamic array, separated with field marks (@FM). This syntax returns the same data as above, but into a single variable:

```
ADDRESS = ""
EXEC SQL SELECT CITY, ST INTO :ADDRESS
    FROM CUSTOMER WHERE ST = "WA"
END EXEC
```

If you do specify multiple variables, values out of the columns are assigned to each variable in turn. The value from the first column is assigned to the first variable, the second value to the second variable, etc. If there are more columns than variables, the last variable becomes a field-mark-delimited array with the remaining column values.

Using Cursors

A majority of the statements you can use in embedded SQL make use of cursors. A cursor is a logical, temporary table that you can access directly within your R/BASIC program.

A cursor is created by executing the equivalent of an SQL SELECT statement. This loads a cursor with a set of columns and rows that meet criteria that you have established.

This is really no different than using interactive SQL. The distinction is that the rows and columns are not simply listed to the screen or printer. Instead, they are placed in the cursor, and your program is given access to them. From that point, you can read the rows or update them.

If you are already familiar with R/BASIC programming, you can compare embedded SQL cursors with select lists. In the case of SQL you must first declare a cursor. This is a slight difference from using select lists in R/BASIC. The declare function simply establishes for SQL the table to be queried and any reduction or sort criteria that will be applicable.

Once you have declared a cursor, you initialize it. In SQL, this is done using the embedded SQL OPEN statement, which actually executes the SQL statement declared earlier with an embedded SQL DECLARE statement. If you were using a select list and non-SQL R/BASIC commands, you would initialize the select list using an R/BASIC SELECT statement.

Information in the cursor can be extracted (FETCH in SQL, READNEXT and READ in R/BASIC). You can also update using the cursor (UPDATE and DELETE in SQL, WRITE and DELETE in R/BASIC).

You can declare any number of cursors in your program. However, you can have only six cursors open at a time. In order to keep track of your cursors, give them a name when you first initialize them.

Cursor Position Embedded SQL maintains information about your position in a cursor. For example, when you first open a cursor, you are positioned before the first row. As you fetch rows out of the cursor, your position is updated, and you move forward in the table.

Normally, you move forward in a cursor until you have processed all the rows in it. However, you can position yourself anywhere within a cursor. This capability is termed “scrolled cursors,” and is controlled by options in the FETCH statement.

Your position with respect to a row is said to be “on,” “before,” or “after” a row. In addition, position can be absolute or relative. Absolute positions include the first or last row. Relative positions include the next or prior row.

Accessing Cursors

Embedded SQL statements that pertain to cursors are generally used in a standard sequence. A typical sequence is illustrated in the following example program (the program is deliberately very simple). In the example, all rows in which the SOURCE column is null are updated with data provided by the user.

```

DECLARE SUBROUTINE MSG

EQU TRUE$ TO 1
EQU FALSE$ TO 0

* macro - cause GOSUB if SQLSTATE = 100
EXEC SQL WHENEVER NOT FOUND GOSUB TRANS_DONE;

* declare cursor name and conditions
EXEC SQL DECLARE TEST_CURSOR CURSOR FOR
      SELECT * FROM CUSTOMERS WHERE SOURCE IS
      NULL
END EXEC

* open (initialize) cursor and execute
* selection conditions
EXEC SQL OPEN TEST_CURSOR;

EXEC SQL BEGIN TRANSACTION;

DONE = FALSE$
LOOP
  * fetch contents of row to variable RECORD
  RECORD = ""
  EXEC SQL FETCH TEST_CURSOR INTO :RECORD ;
UNTIL DONE
  KEY = RECORD<1>
  NEW_VALUE = ""
  MSG("Enter a new value for column F1 of ↵
      row ":KEY,'R',NEW_VALUE,')
  IF NEW_VALUE THEN
    * update table via cursor update
    EXEC SQL UPDATE CUSTOMER
      SET SOURCE = :NEW_VALUE
      WHERE CURRENT OF TEST_CURSOR
    END EXEC
  END
REPEAT

* close cursor
EXEC SQL CLOSE TEST_CURSOR ;
EXEC SQL COMMIT WORK ;
STOP

```

```
TRANS_DONE :
DONE = TRUE$
RETURN
```

The DECLARE statement names the cursor. Using a query expression, DECLARE also establishes the selection and ordering criteria for the rows and columns to be loaded into the cursor. The DECLARE statement does not actually create the cursor — it simply names the cursor and its selection and sort attributes.

The OPEN statement actually executes the query expression specified earlier in a DECLARE statement. After the OPEN statement (assuming no errors), the cursor contains the columns and rows specified in the DECLARE statement.

Once the cursor has been opened, the columns and rows are available to your program. If you wish to read them, use a FETCH statement to put them into a target parameter (R/BASIC variable) for processing.

You can also update information using the cursor. If you use an update statement (UPDATE or DELETE), embedded SQL uses information about the cursor, and your position in it, to update the table from which the cursor was drawn.

When you are finished with a cursor, you must explicitly close it. This differs from using files in R/BASIC (Advanced Revelation files do not need to be explicitly closed in R/BASIC). Until you close a cursor, it is considered active, even after your program quits. Therefore, if you do not close them, you will run out of cursors quickly.

Updating with Cursors

There are restrictions on using a cursor to update a table. This affects the cursor versions of UPDATE and DELETE.

If you initialized a cursor using one of the following conditions, you cannot use the cursor for updates:

- The query expression includes a subquery or a UNION
- The query expression is against a non-updateable View
- The query expression includes ORDER BY
- The query expression includes HAVING
- The query expression includes GROUP BY
- The query expression includes the functions as AVG, SUM, MIN and MAX
- The query expression uses the keyword DISTINCT

Using Cursors in R/BASIC Subroutines

As a rule, you should only declare and open cursors in an R/BASIC main program. However, you can access cursors from within an R/BASIC external subroutine.

In order to share a cursor between a main program and a subroutine, you must use the embedded SQL statement `USING MODULE`. This statement establishes a link between the subroutine and the main program, and allows you to access a cursor initialized in the main program.

The syntax for the `USING MODULE` statement is:

```
EXEC SQL USING MODULE main_program_name
```

in which *main_program_name* is the name of the main program in which the cursor has been initialized.

The `USING MODULE` statement appears in the subroutine *before* you use any other embedded SQL statements. For example, here is a very simple subroutine that fetches rows out of a cursor initialized in the main program called `SQL_MAIN`:

```
SUBROUTINE SQL_SUB(RECORD)
EXEC SQL USING MODULE SQL_MAIN ;
EXEC SQL FETCH TEST_CURSOR INTO :RECORD ;
RETURN
```

It is assumed in this example that the program `SQL_MAIN` contains the proper `DECLARE` and `OPEN` statements to initialize the cursor `TEST_CURSOR`. The subroutine uses the `USING MODULE` statement to “link” itself with the main program, and then uses a `FETCH` statement to return a row from the cursor into a variable. In this instance, the variable is passed back to the main program.

Testing Execution Status (the `SQLSTATE` Variable)

Embedded SQL statements do not directly support a `THEN` or `ELSE` branch. Instead, every embedded SQL statement returns information about its status using the special variable `SQLSTATE`.

`SQLSTATE` is used to indicate both successful operations as well as error conditions. For example, if a `FETCH` statement successfully returns a row from a cursor, `SQLSTATE` is set to zero. On the other hand, if a `FETCH` statement cannot return a value (because the cursor has been exhausted, for example), `SQLSTATE` returns a value of 100. Other error conditions return different values in `SQLSTATE`.

Therefore, in order to get the effect of a `THEN` or `ELSE` branch on an embedded SQL statement, you should test the value of `SQLSTATE` after each embedded SQL statement. The following example illustrates how you might test this value after a `FETCH` statement:

```
DONE = 0
RECORD = ""
LOOP UNTIL DONE
    EXEC SQL FETCH CURSOR_1 INTO :RECORD ;
    IF SQLSTATE THEN
        IF SQLSTATE = 100 THEN DONE = 1
    END ELSE
        GOSUB ERROR
    END
REPEAT
```

There is a great range of values that are possible for SQLSTATE. In general, zero means that the operation was successful, positive values provide information about normal termination, and negative values indicate error conditions. The following table lists common values that you will encounter in using embedded SQL statements:

SQLSTATE Value	Meaning
0	Operation successful
100	No more rows returned
101	All rows returned in SELECT
102	Rows returned, more waiting for SELECT
-104	No more cursors available in OPEN
-300	Logical write error in INSERT
-302	File cannot be opened

You can find the full range of SQLSTATE values listed in a file called INCLUDE. Any record of the name SQLSTATES_ *nnn* (where *nnn* is a number) contains a list of SQLSTATE error explanations. These records are equates (EQU), and are intended to help you use mnemonic names for the possible values in SQLSTATE.

Testing SQLSTATE Automatically

Certain tests of SQLSTATE are so common that embedded SQL provides an automatic way to do them. This is done using the WHENEVER keyword.

WHENEVER establishes default actions to be performed when SQLSTATE returns with one of two values:

Value	Meaning
< 0	Error condition of any kind
100	No more rows to fetch

You would normally have to test for these values using logic similar to that illustrated above.

Instead, however, you can use a WHENEVER clause at the top of your program. After a WHENEVER clause has been executed, embedded SQL checks the value of SQLSTATE after each embedded SQL operation, and acts if it detects the values listed above.

For example:


```
EXEC SQL WHENEVER NOTFOUND GOSUB DONE;
EXEC SQL WHENEVER SQLERROR GOSUB SQL_ERROR;
```

causes the error block in the following example to be created automatically at compile time:

```
EXEC SQL FETCH CUST_CURSOR INTO :RECORD;

* the following error block is
* automatically compiled by WHENEVER
IF SQLSTATE = 100 THEN GOSUB DONE
IF SQLSTATE < 0 THEN GOSUB ERROR
* end of error block
```

Full details on the use of **WHENEVER** appear in the “SQL Reference” chapter.

Compiling Programs

You can compile a program with embedded SQL statements as you would any other R/BASIC program. Use the editor softkey, the Compile window, or the TCL **COMPILE** command.

There is a difference in what the compiler actually accomplishes, though. In addition to compiling the R/BASIC code in your program, the compiler generates and compiles several additional programs.

To state it simply, the compiler generates an SQL module out of your embedded SQL statements. This program in turn then generates additional R/BASIC programs, which are finally compiled into executable subroutines.

The full set of steps accomplished during the compilation of an embedded SQL program is this (in addition to compiling your program):

1. Source code for an SQL module is generated. This module is named **SQLMOD_name**, where *name* is the key to your source code record.
2. An intermediary version of your program is created. In this version, your **EXEC SQL** statements are replaced with **CALL** statements to the SQL module. The intermediary version of your program is called **SQLSRC_name**, where *name* is your program.
3. The SQL module generates corresponding R/BASIC code. This is stored under the name **#SQLMOD_name**.
4. The R/BASIC code for the SQL module is compiled into object code (in other words, **#SQLMOD_name** is compiled).

5. Depending on the actual embedded SQL statements you have used, one or more routines are generated and placed in the table SQL_OBJ. These are named SQLMODE_name_PROC_n_xx, in which *name* is your program, *n* is a number, and *xx* is a suffix (such as "M1") that SQL uses to identify the purpose of the routine.
6. The routine(s) in SQL_OBJ are compiled into R/BASIC code, also stored in the table SQL_OBJ. Unlike the usual R/BASIC programs, the compiled routines are stored with the prefix \$\$\$ (instead of just \$).
7. The SQL module is cataloged if it was not cataloged before.

Not all records created during this process are retained. The intermediary R/BASIC source code records (steps 2, 3, 5) are discarded after the object code is compiled. After a successful compilation, a single R/BASIC program with embedded SQL statements will have created these records:

- Source, object, and symbol table for your program
- The SQL module source, and corresponding R/BASIC object and symbol tables for the SQL module
- Object code records in the table SQL_OBJ for any supporting routines
- A catalog entry for the SQL module

Except for the routines in SQL_OBJ, all generated records appear in the file where your source code resides. The catalog entry is placed in the VOC file.

Compiling only R/BASIC Code

The process described above happens each time you compile an R/BASIC program with embedded SQL statements in it. This is the case even if you make no changes to the embedded SQL portions of your program.

You can compile your programs faster by skipping the additional work required for embedded SQL. To do this, use the TCL command COMPILE, and add the "B" option. For example, the command:

```
COMPILE BP QUERY.REPORT (B)
```

causes the compiler to compile only the R/BASIC portion of your program, and to skip the embedded portion. Of course, if you have made changes that affect the embedded SQL statements in your program, you should recompile without this option.

Capturing Source Code

Once your program with embedded SQL statements has been compiled, you will have two source code records: your program, and the source code for the SQL module. If you wish to capture the additional intermediary R/BASIC source code records, you can use the X option of the compiler at TCL. For example:

```
COMPILE BP QUERY.REPORT (X)
```

prevents the compiler from deleting the records
 SQLSRC_QUERY.REPORT and
 #SQLMOD_QUERY.REPORT when the object code has been
 filed, and the records
 SQLMOD_QUERY.REPORT_PROC_n_xx are in SYS_OBJ.

Transaction Processing and Embedded SQL

You can control transaction processing from within embedded SQL using the same statements you use in interactive SQL. These are the BEGIN TRANSACTION, SET TRANSACTION, COMMIT WORK and ROLLBACK WORK statements.

When the COMMIT WORK or ROLLBACK WORK statement is executed, all cursors are closed. A ROLLBACK WORK statement also unlocks any records or files that have been implicitly locked during the transaction (it does not unlock locks that you set explicitly using the R/BASIC LOCK command). The COMMIT WORK statement will unlock *all* locks if the environment setting *Unlock All at Commit* has been set.

- For more information, see the “Managing Files and Records” chapter in *Filing Systems*.



Chapter K8

SQL Reference

This chapter provides the technical description of Advanced Revelation SQL.

- Extensions to SQLK 8 . 1
- Reserved WordsK 8 . 4
- Advanced Revelation SQL Data TypesK 8 . 5
- SQL OperatorsK 8 . 6
- SQL KeywordsK 8 . 7
- SQL StatementsK 8 . 30



Extensions to SQL

Advanced Revelation SQL is a superset of the ANSI level 2 SQL specification (SQL2), including special Advanced Revelation extensions.

The extensions provided in Advanced Revelation are intended to:

- Provide a complete, rigorous SQL2 implementation
- Enable you to take advantage of Advanced Revelation tools and architecture

The issues addressed by the Advanced Revelation extensions will be familiar to most SQL users.

Assigning Primary Keys

Most implementations of SQL do not require primary keys. This permits ambiguity and conflicting references, particularly with regard to foreign keys (references to joined tables).

Advanced Revelation SQL resolves this with the PRIMARY KEY keyword, used with the CREATE TABLE statement. This extension requires you to designate a column(s) as the primary key for each table. PRIMARY KEY requires that each value for the column is both not null and unique (concurrent use of the NOT NULL and UNIQUE keywords is redundant).

Documenting References

SQL does not provide for updating a foreign key as data is changed; changes are made only to a single table. You are then responsible for ensuring that the changes are reflected for relevant columns in another table.

The REFERENCES keyword for the CREATE TABLE or ALTER TABLE statements helps to track foreign key references by giving you the option of listing the tables in which a column is a foreign key.

Controlling Concurrent Transactions

Although standard SQL provides transaction processing, it does not define concurrency controls. For networks or multiuser systems, it is important that you have a way to regulate conflicting transactions.

In these circumstances, reliable data is affected by competing needs: access and protection. By adjusting the balance between each, you can maintain data integrity without sacrificing efficient use.

The SET TRANSACTION statement gives you a range of locking controls. You can adjust these settings from transaction to transaction.

Changing a Table Definition

Many SQL implementations do not give you a way to alter the definition of a table. The standard operations enable you to insert and delete rows, or to simply delete the table and start over.

Using the ALTER TABLE statement in Advanced Revelation SQL, you can add, modify, or delete column definitions. All of the column specifications available to you when you create a table can also be used when you alter a table.

⚠ However, you cannot add or delete primary key columns.

Allowing Repeating Data

Repeating data (columns having multiple values) is handled in SQL through the normalization of data: using separate rows for each value.

The Advanced Revelation architecture incorporates multiple values for fields. By including the REPEATING option, you can have full access to Advanced Revelation files that have 'multivalued' fields.

The REPEATING keyword is used with the CREATE TABLE or ALTER TABLE statements to define a column as multivalued.

*Table K8-1***SQL2 Proposed Extensions**

Statement	Keywords
ALTER TABLE	ADD CURRENT DEFAULT DROP MODIFY REFERENCES USER
CREATE TABLE	CURRENT DEFAULT PRIMARY KEY REFERENCES USER
SET TRANSACTION	CONSISTENCY LEVEL READ ONLY READ WRITE

*Table K8-2***Advanced Revelation SQL Keyword Extensions**

Statement	Advanced Revelation Keyword
ALTER TABLE, CREATE TABLE	REPEATING ON
SELECT	SUPPRESS DISPLAY PRINTER

Reserved Words

Most of the reserved words for Advanced Revelation SQL are defined as such by the current ANSI SQL standard. None of these words may be used as identifiers in SQL statements.

Table K8-3

Advanced Revelation SQL Reserved Words

ADD	FOUND	REFERENCES
ALL	FROM	REPEATING
AND	GO	ROLLBACK
ANY	GOSUB	SCHEMA
AS	GOTO	SECTION
ASC	GRANT	SELECT
AUTHORIZATION	GROUP	SET
AVG	HAVING	SQL
BEGIN	IN	SQLCODE
BETWEEN	INDICATOR	SQLERROR
BY	INSERT	SUM
CHAR	INT	TABLE
CHARACTER	INTEGER	TO
CHECK	INTO	TRANSACTION
CLOSE	IS	UNION
COMMIT	LANGUAGE	UNIQUE
CONSISTENCY LEVEL	LIKE	UPDATE
CONTINUE	MAX	USER
COUNT	MIN	VALUES
CREATE	MODIFY	VARCHAR
CURRENT	MODULE	VIEW
CURSOR	NOT	WHENEVER
DEC	NULL	WHERE
DECIMAL	NUMERIC	WITH
DECLARE	ON	WORK
DEFAULT	OPEN	
DELETE	OPTION	
DESC	OR	
DISTINCT	ORDER	
DOUBLE	PRECISION	
DROP	PRIMARY KEY	
END	PRINTER	
ESCAPE	PRIVILEGES	
EXEC	PROCEDURE	
EXISTS	PUBLIC	
FETCH	READ ONLY	
FLOAT	READ WRITE	
FOR	REAL	

Advanced Revelation SQL Data Types

The following table lists the data types and associated keywords for ANSI standard SQL. A full listing of data types supported in Advanced Revelation is provided in the chapter “Advanced Revelation Files and Records” in *Filing Systems*.

Information regarding the use of a particular keyword is available under the topic “SQL Keywords”.

Table K8-4

Advanced Revelation SQL Data Types

Keyword	Usage	Examples
CHAR(<i>n</i>)	Fixed-length to <i>n</i> characters	Kipp, 123, #1
VARCHAR	Variable-length to 255 characters	Advanced Revelation, 1989, **
TEXT	Variable-length no limit	If @ANS = “Yes”
INTEGER	Integers	0, -4, 123
DECIMAL(<i>n,m</i>)	Fixed-point to <i>n, m</i> digits	0, 14.55, .25
FLOAT	Floating-point	0, 2.18693
DOLLARS	Monetary	\$22.95
DATE	Date	1 April 1988
DATETIME	Date and Time	1/4/88 08:00:00
TIME	Time	11:30:00
BOOLEAN	Boolean values	0/1, true/false

SQL Operators and Notation

Table K8-5
SQL Operators

Type	Operator	Meaning
Math	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
Comparison	=	Equals
	< >	Not Equal
	<	Less Than
	< =	Less Than/Equal
	>	Greater Than
	> =	Greater Than/Equal
Boolean	AND	A logical AND operation
	OR	A logical OR operation
	NOT	A logical NOT operation

Table K8-6
SQL Notation

Type	Operator	Meaning
Syntax	;	Delimits embedded SQL statements
	*	All columns
	()	Encloses function arguments, multiple column definitions or assignments, column equations, and nested queries
	%	(percent) Wildcard character matching any combination of zero or more characters. Used with LIKE keyword.
	_	(underscore) Wildcard character matching a single character. Used with LIKE keyword.
	—	(two dashes) Precedes any comments in embedded SQL code.

SQL Keywords

The keywords are used with SQL statements. Some of the keywords are merely placeholders, lending more clarity to a statement and introducing a parameter. Other keywords operate as functions or procedures, or designate data types and system values.

This section describes how and where keywords are used, giving examples for each.

ADD	K 8 . 10
ALL	K 8 . 10
ANY	K 8 . 11
AS	K 8 . 11
AVG	K 8 . 11
BETWEEN	K 8 . 11
CONSISTENCY LEVEL	K 8 . 12
CONTINUE	K 8 . 13
COUNT	K 8 . 13
COUNT(*)	K 8 . 13
CURRENT	K 8 . 13
CURRENT OF	K 8 . 14
CURSOR FOR	K 8 . 14
DEFAULT	K 8 . 14
DISTINCT	K 8 . 14
DROP	K 8 . 15
EXISTS	K 8 . 15
FROM	K 8 . 15
GOSUB	K 8 . 15
GOTO	K 8 . 16
GROUP BY	K 8 . 16

HAVING	K 8 . 16
IN	K 8 . 17
INTO	K 8 . 17
KEYQUAL	K 8 . 17
LIKE	K 8 . 18
MAX	K 8 . 18
MIN	K 8 . 19
MODIFY	K 8 . 19
NOTFOUND	K 8 . 19
NULL	K 8 . 20
ON	K 8 . 20
ORDER BY	K 8 . 21
PRIMARY KEY	K 8 . 22
REFERENCES	K 8 . 22
REPEATING	K 8 . 22
SET	K 8 . 23
SOME	K 8 . 23
SQLERROR	K 8 . 23
SUM	K 8 . 23
SUPPRESS DISPLAY	K 8 . 24
TO PRINTER	K 8 . 24
UNIQUE	K 8 . 24
UNION	K 8 . 24
USER	K 8 . 25
VALUES	K 8 . 25

WHENEVERK 8 . 26

WHEREK 8 . 26

WITH CHECK OPTIONK 8 . 29

WORKK 8 . 29

ADD

Used with the **ALTER TABLE** statement to add a column definition to a table. The statement

```
ALTER TABLE CUSTOMERS ADD ADDRESS VARCHAR
```

adds a new **CONTACT** column definition to the table of clients.

ALL

Used with the **SELECT** statement, and in the **UNION** and **WHERE** clauses.

In **SELECT** statements, **ALL** indicates that the **SELECT** operation should return all rows selected, whether there are duplicate rows or not. Since this is the default **SELECT** mode, **ALL** may be specified for clarity.

☛ See the **DISTINCT** keyword.

In **UNION** clauses, **ALL** specifies that the results table produced by the union of tables should contain all rows. The default mode for **UNION** is to use only distinct rows, ignoring any duplicates.

☛ See the **UNION** keyword.

As a comparison keyword in a **WHERE** clause, **ALL** is preceded by one of the comparison operators (such as **<**, **>**, **=**) and followed by a subquery. **ALL** indicates that the operation must be successful against all values returned by the subquery. The result of the **ALL** operation is determined by the operator used.

For example, the query

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
WHERE PRICE > ALL
(SELECT PRICE FROM PRODUCTS_88)
```

returns the description and price of the product having a higher price than any last year.

ANY

A comparison keyword used in a WHERE clause. The keywords ANY and SOME are identical in operation. They are preceded by one of the operators (such as <, >, =) and followed by a subquery. Both keywords specify that the operation must be successful against any one of the values returned by the subquery. Each value within range of the comparison operator is returned.

The query,

```
SELECT CODE, QUANTITY FROM INVOICES_88
WHERE QUANTITY > ANY
(SELECT PRICE FROM INVOICES)
```

returns the code and quantity of the product that sold more last year than this year.

AS

Used with the CREATE VIEW statement to introduce the query defining the view. For example:

```
CREATE VIEW CUSTOMER_CONTACT AS
SELECT LAST_NAME, FIRST_NAME, PHONE FROM CUSTOMERS
```

AVG

A function keyword used with the SELECT statement. AVG divides the sum of the selected columns by the number of columns selected. The query,

```
SELECT AVG(PRICE) FROM PRODUCTS
```

returns the average price for the products.

BETWEEN

A comparison keyword used in a WHERE clause. BETWEEN compares the value of the column for inclusion within a range of values. The query,

```
SELECT CODE, PRICE FROM INVOICES, PRODUCTS
WHERE (QUANTITY * PRODUCT.PRICE) BETWEEN 50.00 AND 100.00
```

will return the code for the products having orders greater than or equal to \$50.00 and less than or equal to \$100.00.

CONSISTENCY LEVEL

Used with the SET TRANSACTION statement to specify the type and duration of locking for transactions. The CONSISTENCY LEVEL settings must be specified as the first statement of a transaction.

- ☛ The level specified overrides the *Consistency Level* setting at the Concurrency Environment window.

The CONSISTENCY LEVEL keyword is followed by a number specifying one of the available level settings:

- 0 No implicit locking is done.
- 1 Enables exclusive locking during updates. The lock is held for the duration of the update.
- 2 Enables exclusive locking during updates. The lock is held for the duration of each transaction.
- 3 Enables exclusive locking during updates for the duration of each transaction, and shared locking during queries for the duration of the query.
- 4 Enables exclusive table locking during updates and shared table locking during queries. The locks are held for the duration of each transaction.

The *Implicit Locking* setting at the Global Environment window must be set to File locking in order to use this level.

A level 4 setting guarantees serializable transactions when *Commit Protection* is set at the Global Environment window.

For example,

```
SET TRANSACTION CONSISTENCY LEVEL 3
```

will enable shared locking. The lock is released at the end of the update.

- ☛ For more information, see “Locking” chapter in the *Networks* section.

CONTINUE

Used in embedded SQL in a **WHENEVER** clause. **CONTINUE** causes a module to continue execution after the **WHENEVER** test is true for an error condition. For example,

```
EXEC SQL WHENEVER SQLERROR CONTINUE
```

stipulates that if **SQLERROR** is true, execution of the statements in the module will still continue.

COUNT

A function keyword used with the **SELECT** statement to count the number of rows having non-null values. The query,

```
SELECT COUNT(DISTINCT CODE) FROM INVOICES  
WHERE QUANTITY > 1
```

returns the number of distinct products that have more than one ordered.

COUNT(*)

A function keyword used with the **SELECT** statement to count all rows selected. Rows having null or duplicate values are included in the count. The query,

```
SELECT COUNT(*) FROM INVOICES  
WHERE QUANTITY > 1
```

returns the number of products that have more than one ordered. Rows having a null or duplicate value are included in the total.

CURRENT

Used with the **DEFAULT** keyword in the **CREATE TABLE** and **ALTER TABLE** statements. **CURRENT** denotes the current date and time, inserted if an explicit value is not assigned. For example,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, DATE DATETIME  
DEFAULT CURRENT)
```

inserts the current datetime as the default value of the **DATE** column.

CURRENT OF

Used with the WHERE keyword in embedded DELETE and UPDATE statements to reference a variable assigned to a cursor. For example,

```
DELETE FROM CUSTOMERS WHERE CURRENT OF CURSOR1
```

The row at the current CURSOR1 position is deleted.

CURSOR FOR

Used with the embedded DECLARE statement to reference a query to be associated with a cursor variable. For example,

```
DECLARE CURSOR1 CURSOR FOR  
SELECT LAST_NAME FROM CUSTOMERS WHERE CUSTOMER_ID = 330
```

associates the SELECT statement with CURSOR1.

DEFAULT

Used with the CREATE TABLE or ALTER TABLE statement to assign a default value to the column being defined. The DEFAULT keyword is followed by the value to assign. The value may be:

- A literal value
- The NULL keyword
- The CURRENT keyword (inserts current datetime)
- The USER keyword (inserts current user name)

For example,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, DATE DATE  
DEFAULT "30 July 1989")
```

assigns the DATE column a default value of "30 July 1989".

DISTINCT

Used with the SELECT statement. Only one representative row, among any duplicate rows, is selected. For example, the query,

```
SELECT DISTINCT CODE FROM INVOICES
```

returns the codes for the products ordered. A particular code will appear only once, despite the number of times it may appear in the column.

✎ DISTINCT may not be used with the function keyword COUNT(*).

DROP

Used with the ALTER TABLE statement to delete a column from a table. For example,

```
ALTER TABLE INVOICES DROP CODE
```

deletes the CODE column definition and any data in this column from the INVOICES table.

EXISTS

Used in a WHERE clause to test for the existence of specific data. A negative test is formed by preceding EXISTS with the NOT boolean operator. The query,

```
SELECT PRODUCT_CODE, DESCRIPTION FROM PRODUCTS
WHERE EXISTS
  (SELECT * FROM INVOICES
   WHERE DATE = "22 October 1989")
```

returns the code and description for any products on the specified date.

FROM

Used with the SELECT and DELETE statements to precede the name of a table(s). The statement,

```
DELETE * FROM INVOICES
WHERE DATE = "13 January 1989"
```

removes any orders from the INVOICES table entered on the specified date.

GOSUB

Used in embedded SQL in a WHENEVER clause. The GOSUB clause is executed if the WHENEVER test evaluates to true. GOSUB passes execution to a program subroutine. The format is:

```
GOSUB subroutine
```

For example,

```
EXEC SQL WHENEVER NOTFOUND GOSUB ERROR_CONDITION
```

will transfer execution to the ERROR_CONDITION subroutine if NOTFOUND is true.

GOTO

Used in embedded SQL in a **WHENEVER** clause. The **GOTO** clause is executed if the **WHENEVER** test evaluates to true. **GOTO** passes execution to the program statement following the referenced label identifier. The format is:

```
GOTO label
```

For example,

```
EXEC SQL WHENEVER NOTFOUND GOTO NEXT_MODULE
```

will transfer execution to the program statement following the **NEXT_MODULE** label if **NOTFOUND** is true.

GROUP BY

Used with the **SELECT** statement to group selected rows. Each group of rows can then be operated upon as a unit. For example, the query,

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
GROUP BY PRICE
ORDER BY DESCRIPTION
```

returns the product descriptions and price for all products. The rows are grouped according to the price, and each product group is sorted by description.

HAVING

Used with the **SELECT** statement following a **GROUP BY** clause. The **HAVING** operation applies its search conditions to groups of rows as a whole: each entire group must meet the conditions. If the conditions are met, the group is selected. When it is not preceded by a **GROUP BY** clause, a **HAVING** operation will test the full table as a group.

The **HAVING** clause may test whether specific data exists, or specify conditions to be met. See the **WHERE** keyword for descriptions of the parameters.

The syntax is identical to that of a **WHERE** clause:

```
HAVING expression comparison subquery {boolean} ::
expression
```

For example, the query,

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
GROUP BY PRICE
HAVING PRICE > 30.00
ORDER BY DESCRIPTION
```

returns the product descriptions and price for all products priced equal to or higher than \$30.00. The rows are grouped according to the price, and each product group is sorted by description.

IN

A comparison keyword used in a WHERE clause. IN tests the value of a column for inclusion in a list of values. For example, the query,

```
SELECT CUSTOMER_ID, LAST_NAME, FIRST_NAME FROM CUSTOMERS
WHERE LAST_NAME IN ("Zander", "Smith", "Thomas")
```

checks each row for a last name that appears in the list. If any of the three names are found, the customer id, last and first names are returned.

INTO

Used with the INSERT and embedded SELECT statement. When used with the INSERT statement, INTO introduces the name of a table storing the row. When used in an embedded SELECT statement, INTO references the local variable(s) storing the row.

For example,

```
INSERT INTO INVOICES (CODE, QUANTITY) VALUES (010101, 4)
WHERE CUSTOMER_CODE = 330
```

KEYQUAL

The name of the column created automatically when a multivalued column(s) is defined for the table. The KEYQUAL values are indices into the multivalued columns.

If a KEYQUAL value is not specified when a row is added to the table, a sequential value is automatically assigned.

For example, the statement:

```
INSERT INTO INVOICES (CODE, QUANTITY, KEYQUAL) VALUES  
(030100, 1, 2) WHERE CUSTOMER_ID = 330
```

may be rewritten as:

```
INSERT INTO INVOICES (CODE, QUANTITY) VALUES (030100, 1)  
WHERE CUSTOMER_ID = 330
```

to allow automatic assignment of the **KEYQUAL** value.

LIKE

A comparison keyword used in a **WHERE** clause to compare the value of the column in the row being tested with a string. **LIKE** employs the characters % and _ as wildcard characters:

%	Matches any combination of character(s)
_	Matches any single character

For example, the query

```
SELECT CUSTOMER_ID, LAST_NAME, FIRST_NAME FROM CUSTOMERS  
WHERE LAST_NAME LIKE "Ing%m_"
```

will check the **LAST_NAME** column of the **CUSTOMERS** table for a name that begins with 'Ing', any number of subsequent letters, 'm', and any single following letter.

The query,

```
SELECT PRODUCT CODE, DESCRIPTION, PRICE FROM PRODUCTS  
WHERE DESCRIPTION LIKE "Navy%"
```

returns any product descriptions that begin with 'Navy' and is followed by any number of letters.

MAX

A function keyword used with the **SELECT** statement to find the maximum column value from the selected rows. For example, the query,

```
SELECT CODE, MAX(QUANTITY) FROM INVOICES
```

checks for the largest order for a product and returns the product code and number ordered.

The query,

```
SELECT CODE, MAX(QUANTITY * PRICE) FROM INVOICES, PRODUCTS
```

checks for the largest invoice amount and returns the product code and amount.

MIN

A function keyword used with the SELECT statement to find the minimum value in the selected columns. The query,

```
SELECT CODE, MIN(QUANTITY * PRICE) FROM INVOICES, PRODUCTS
```

checks for the smallest invoice amount and returns the product code and amount.

MODIFY

Used with the ALTER TABLE statement to change the definition of an existing column. For example, the statement,

```
ALTER TABLE CUSTOMERS  
  MODIFY PHONE CHAR(12) REPEATING
```

changes the column specification for PHONE. The column definition is changed from a single valued column to a multivalued column (REPEATING).

NOTFOUND

Used in embedded SQL in a WHENEVER clause. NOTFOUND may be either true or false. If NOTFOUND evaluates to true, a specified action is performed. NOTFOUND is equivalent to an SQLSTATE value of 100.

- ☛ For more information, see the system variable SQLSTATE in the chapter “Programming with SQL”.

For example,

```
EXEC SQL WHENEVER NOTFOUND GOSUB DONE
```

will transfer execution to the DONE subroutine if NOTFOUND is true.

NULL

A keyword used with the ALTER TABLE and CREATE TABLE statements, and in a WHERE clause.

In WHERE clauses, IS (NOT) NULL compares the value of the column with a null value. For example, the query,

```
SELECT NAMES FROM APPLICANTS
WHERE CREDIT_HISTORY IS NOT NULL
```

returns the names of all applicants having a credit record of any sort.

In the ALTER TABLE and CREATE TABLE statements, NULL is used:

- In a DEFAULT clause to assign a null value to a column
- With the NOT operator to specify that the column may not be assigned a null value

For example, the statement,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, DATE DATE
DEFAULT NULL)
```

assigns a default null value to the DATE column.

The statement,

```
ALTER TABLE INVOICES
MODIFY QUANTITY INTEGER NOT NULL
```

specifies that the START_DATE column is changed to reject any null values.

ON

An Advanced Revelation extension. Used with the CREATE TABLE statement to assign the table to a particular Advanced Revelation volume. The table created with the statement

```
CREATE TABLE CUSTOMERS ON DATAVOL
```

will exist on the volume DATAVOL.

- ✎ For more information, see the “Advanced Revelation Volumes” chapter in *Filing Systems*.

ORDER BY

Used with the **SELECT** statement to sort the rows selected according to the value of the specified columns. The rows may be sorted in ascending or descending order.

The syntax of an **ORDER BY** clause is:

```
ORDER BY column order
```

The ordering keywords **ASC** or **DESC** indicate an ascending (lowest value first) or descending (highest first) sort. The *order* parameter defaults to **ASC**.

For example,

```
SELECT ST FROM CUSTOMERS
ORDER BY ST
```

will alphabetize a listing of the states in the **CUSTOMERS** table.

The query,

```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
ORDER BY PRICE DESC
```

will order the description and price information returned from highest to lowest price.

You can specify any number of columns to sort. For example,

```
SELECT LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST, ZIP FROM
CUSTOMERS
WHERE ST = "AZ"
ORDER BY CITY, ZIP, LAST_NAME
```

sorts the customers in Arizona by city, then by zip code within each city, and by last name within each zip code area.

In addition, you can use the number rather than the name for a corresponding column. The previous example might be rewritten as:

```
SELECT LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST, ZIP FROM
CUSTOMERS
WHERE ST = "AZ"
ORDER BY 4, 6, 1
```

where **CITY** is the fourth column, **ZIP** is the sixth column, and **LAST_NAME** is the first column in the **CUSTOMERS** table.

{PRIMARY} KEY

PRIMARY is an optional specifier, but is supplied for common usage. **KEY** is used in column definitions with the **CREATE TABLE** or **ALTER TABLE** statements to define a column as a primary key for a row. **KEY** implies that each value for the column is unique and not null (concurrent use of the **NOT NULL** and **UNIQUE** keywords is redundant).

✎ Advanced Revelation SQL requires that all rows have a primary key.

For example,

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER KEY)
```

assigns the value of **CUSTOMER_ID** as the primary key for each row in the **CUSTOMERS** table.

REFERENCES

REFERENCES is used with the **CREATE TABLE** or **ALTER TABLE** statements to document a column as a foreign key. The **REFERENCES** keyword is followed by a list of tables (comma-delimited). For example,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, CUSTOMER_ID IN-  
TEGER NOT NULL UNIQUE REFERENCES CUSTOMERS)
```

The **CUSTOMER_ID** column is a foreign key referencing the **CUSTOMERS** table.

✎ The foreign key references are displayed in the dictionary listing for the **CUSTOMERS** table.

REPEATING

An Advanced Revelation extension. **REPEATING** is used with the **CREATE TABLE** and **ALTER TABLE** statements to define a column as multivalued (having multiple values). For example, the statement,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, QUANTITY  
INTEGER REPEATING)
```

defines the **QUANTITY** column as a multivalued column of integers.

SET

Used with the **UPDATE** statement to introduce a new column assignment(s) for the column(s) being updated. For example,

```
UPDATE CUSTOMERS SET ADDRESS = "831 116th Ave"  
WHERE CUSTOMER_ID = 330
```

adds address information for a particular customer.

SOME

A comparison keyword used in a **WHERE** clause. The keywords **SOME** and **ANY** are identical in operation. They are preceded by one of the operators (such as **<**, **>**, **=**) and followed by a subquery. Both keywords specify that the operation must be successful against some of the values returned by the subquery. Each value within range of the comparison operator is returned. For example, the query,

```
SELECT CODE, QUANTITY FROM INVOICES_88  
WHERE QUANTITY > SOME  
(SELECT PRICE FROM INVOICES)
```

returns the code and quantity of the product that sold more last year than this year.

SQLERROR

Used in embedded SQL in a **WHENEVER** clause. **SQLERROR** may be either true or false. If **NOTFOUND** evaluates to true, a specified action is performed. **SQLERROR** is equivalent to an **SQLSTATE** value of less than zero.

- ☛ For more information, see the system variable **SQLSTATE** in the chapter "Programming with SQL".

For example,

```
EXEC SQL WHENEVER SQLERROR GOTO FLAG
```

will transfer execution to the program statement following the **FLAG** label if **SQLERROR** is true.

SUM

A function keyword used with the **SELECT** statement to total the values for a column or column equation. The query,

```
SELECT SUM(QUANTITY) FROM INVOICES  
WHERE CODE = 020120
```

returns the total number of orders for the product code specified.

SUPPRESS DISPLAY

An Advanced Revelation extension. **SUPPRESS DISPLAY** is an option used with the **SELECT** statement to suppress the display of any rows selected (results table). The results table is instead sent to the **LISTS** file. Data may be retrieved from the **LISTS** file at any time from **TCL** with the **R/LIST** statement **GETLIST**.

☛ For more information, see *R/LIST* or the **GETLIST** statement in *TCL*.

For example,

```
SELECT LAST_NAME, FIRST_NAME, PHONE FROM CUSTOMERS
WHERE STATE = AZ
ORDER BY LAST NAME
SUPPRESS DISPLAY
```

will select all customers in Arizona order them by last name. Rather than being displayed, the results table is written to the **LISTS** file.

TO PRINTER

An Advanced Revelation extension. **TO PRINTER** is an option used with the **SELECT** statement to send the selected rows (results table) to the default printer. For example,

```
SELECT * FROM CUSTOMERS TO PRINTER
```

will output to the printer a results table of all data in the **CUSTOMERS** table.

UNIQUE

Used with the **CREATE TABLE** and **ALTER TABLE** statements to specify that the column may not contain any duplicate values. **SQL** requires that **UNIQUE** is always preceded by **NOT NULL**, since uniqueness is not applicable to null values. For example, the statement,

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, CUSTOMER_ID IN-
TEGER NOT NULL UNIQUE)
```

requires that only unique customer id numbers may be entered for the table.

UNION

Used with the **SELECT** statement to combine the results of two queries to produce a single table. The data type and length of the columns selected from each table must be identical. A results table of the selected rows is returned; any duplicate rows are not included.

For example, the query,

```
SELECT DISTINCT CODE, QUANTITY FROM INVOICES
UNION
SELECT DISTINCT CODE, QUANTITY FROM INVOICES_88
```

will combine the product codes and quantities ordered this year with those of last year.

To include duplicate rows in a union, use the ALL keyword immediately after UNION. The previous example would be rewritten as:

```
SELECT DISTINCT CODE, QUANTITY FROM INVOICES
UNION ALL
SELECT DISTINCT CODE, QUANTITY FROM INVOICES_88
```

Multiple unions can also be performed. In this case, the associated queries are enclosed in parentheses. For example,

```
SELECT * FROM PRODUCTS
UNION
(SELECT * FROM PRODUCTS_88
UNION
SELECT * FROM PRODUCTS_87)
```

combines all product information for the current year and the past two years.

USER

Used with the DEFAULT keyword in CREATE TABLE and ALTER TABLE statements. USER is a SQL system variable having a value of the current user name. For example,

```
CREATE VIEW INVOICE_ENTRY SALESPERSON CHAR(25) DEFAULT USER
```

assigns the SALESPERSON column the default value of the current user name.

VALUES

Used with the INSERT statement to introduce the specified value(s) to be assigned to a column. For example, the statement,

```
INSERT INTO INVOICES (CODE, QUANTITY) VALUES (010101, 4)
WHERE CUSTOMER_ID = 301
```

will insert the value 010101 into the CODE column, and the value 4 into the QUANTITY column for the specified customer.

WHENEVER

Used in embedded SQL to test for error conditions within a module. The WHENEVER clause must appear before any statements in the module. After each statement executes, the WHENEVER test is performed.

The format for a WHENEVER clause is:

```
WHENEVER condition action
```

where *condition* is the status to test. The *condition* may be one of the keywords:

- NOTFOUND
- SQLERROR

Action refers to the operation to be performed if the *condition* evaluates to true. The *action* may be one of the keywords:

- GOSUB *label*
- GOTO *label*
- CONTINUE

For example,

```
EXEC SQL WHENEVER NOTFOUND GOSUB DONE
```

tests the status of the NOTFOUND keyword after each statement in the module executes. When NOTFOUND is true, the GOSUB clause is executed.

- ☛ For more information, see the chapter “Programming with SQL”, and the keywords NOTFOUND, SQLERROR, GOSUB, GOTO, and CONTINUE.

WHERE

Used with SELECT, UPDATE, and DELETE statements to test individual rows at a time for processing. This may be a test of whether specific data exists, or a condition(s) to be met.

The WHERE and HAVING keywords use the same syntax:

```
WHERE expression comparison subquery {boolean} ::  
      expression
```


Parameters

expression An *expression* may be:

- A column or columns (separated by commas).
For example,

```
LAST_NAME, DESCRIPTION
```

- A column equation using any of the function keywords or math operators. The expression is enclosed in parentheses. Some examples are:

```
(QUANTITY * PRICE)
```

```
SUM(column1)
```

```
AVG(column1 + column2)
```

- A literal value. Examples are:

```
"User", 23, "5/21"
```

- An equation using literal values. For example:

```
(31 - 14)
```

- A list of literal values when used as the second *expression* with the IN keyword. For example:

```
IN (Annette, Claire, Helen, 36)
```

comparison A *comparison* may be:

- One of the following comparison keywords:

```
{NOT} EXISTS
```

```
IS {NOT} NULL
```

```
{NOT} BETWEEN
```

```
{NOT} LIKE {ESCAPE character}
```

```
{NOT} IN
```

The {ESCAPE *character*} option with LIKE identifies a literal character to be interpreted as an escape character.

The EXISTS keyword is not preceded by an *expression*. The IS NULL keyword is not followed by an *expression* or *subquery*.

- One of the comparison operators (such as <, >=, =)
- A combination of one of the comparison operators and one of the following comparison keywords:

ALL
ANY
SOME

- One of the boolean operators AND or OR

subquery A *subquery* is a nested query. An example might be:

```
(SELECT LAST_NAME, FIRST_NAME, PHONE FROM
CUSTOMERS)
```

The ALL, ANY, SOME, and EXISTS keywords require a subquery. It is an option for the IN keyword.

boolean When enclosed in parentheses, multiple clauses may be combined using the AND or OR boolean operators, as in:

```
WHERE (QUANTITY) > 0 AND
(AVG(PRICE) > 50.00)
```

Examples

```
SELECT ST FROM CUSTOMERS
  WHERE CITY = "Placentia"
```

```
SELECT LAST_NAME, FIRST_NAME FROM CUSTOMERS
  WHERE LAST_NAME LIKE "Xen%"
```

```
SELECT QUANTITY FROM INVOICES
  WHERE CODE IN (040130, 020120, 030320)
```

```
SELECT PRODUCT_CODE, DESCRIPTION FROM
PRODUCTS
  WHERE EXISTS
    (SELECT * FROM INVOICES
      WHERE CODE = PRODUCT_CODE)
```

WITH CHECK OPTION

Used with the CREATE VIEW statement to specify that any UPDATE or INSERT operations affecting the view must conform to the definition of the view (fall within the 'scope' of the view, as specified in the WHERE clause). For example,

```
CREATE VIEW CUSTOMER_CONTACT AS
  SELECT LAST_NAME, FIRST_NAME, PHONE FROM CUSTOMERS
     WHERE ST = "WA" AND ZIP = 98133
  WITH CHECK OPTION
```

would require that any customer information inserted into the view would have to be for Washington state in the zip code area 98133.

If WITH CHECK OPTION is not used, alterations to a table may affect the function or usefulness of any associated views.

WORK

Used with the COMMIT and ROLLBACK statements for added clarity. WORK has no functional significance. For example,

```
COMMIT WORK
```

SQL STATEMENTS

This section describes, illustrates, and gives the complete syntax for the Advanced Revelation SQL statements.

ALTER TABLE	K 8 . 31
BEGIN TRANSACTION	K 8 . 33
CLOSE (EMBEDDED SQL)	K 8 . 33
COMMIT	K 8 . 34
CREATE TABLE	K 8 . 34
CREATE VIEW	K 8 . 36
DECLARE (EMBEDDED SQL)	K 8 . 38
DELETE	K 8 . 39
DELETE (EMBEDDED SQL)	K 8 . 40
DROP TABLE	K 8 . 41
DROP VIEW	K 8 . 41
FETCH (EMBEDDED SQL)	K 8 . 42
INSERT	K 8 . 44
OPEN (EMBEDDED SQL)	K 8 . 46
ROLLBACK	K 8 . 46
SELECT	K 8 . 47
SELECT (EMBEDDED SQL)	K 8 . 51
SET TRANSACTION	K 8 . 55
UPDATE	K 8 . 57
UPDATE (EMBEDDED SQL)	K 8 . 58

ALTER TABLE

The ALTER TABLE statement adds, modifies, or deletes a column definition from a table or view. A primary key column cannot be modified.

☞ Existing data may be affected by an ALTER TABLE operation on a table in a bonded environment. Refer to the bond documentation.

```
ALTER TABLE table [ 

ADD  
MODIFY  
DROP

 ] column type {DEFAULT value} ⇨  
  
{column_def}
```

table

The name of an existing table or view

ADD

A keyword that adds a new column to a table. The ADD and MODIFY keywords use the same syntax.

MODIFY

A keyword to change the specifications for an existing column.

DROP

A keyword to delete a column from a table. The column definition as well as any data existing in the column are deleted.

column

The name of the column to ADD, MODIFY, or DROP in *table*

type

The data type specification for *column*

☞ See *Table K8-4*, “SQL Data Types” in this chapter for a listing of available data types.

DEFAULT

A keyword that assigns a default value to the column.

value

The value assigned may be:

- A literal value
- The NULL keyword
- The CURRENT keyword (DATETIME)
- The USER keyword (user name)

column_def

Column_def identifies additional definitions that may be applied to a column. More than one of the following keywords may be used (not delimited) in the order listed:

NOT NULL	<i>Column</i> must be assigned a non-null value.
UNIQUE	Used only with the NOT NULL keyword. <i>Column</i> may not contain any duplicate values.
REFERENCES	Column is a foreign key in a specified list of <i>tables</i> . REFERENCES is followed by a <i>tables</i> parameter.
<i>tables</i>	A list of the table(s) to cross reference (delimited by commas).
REPEATING	Column may be assigned multiple values.

Examples

```
ALTER TABLE CUSTOMERS MODIFY ADDRESS VARCHAR NOT NULL
```

Modifies existing ADDRESS column to allow character data of any length and require a non-null value.

```
ALTER TABLE CUSTOMERS ADD ST CHAR(2) DEFAULT "WA"
```

Adds new ST column. The column allows character data of up to 12 characters in length, and defaults to "WA".

```
ALTER TABLE CUSTOMERS ADD SALESPERSON VARCHAR DEFAULT USER
```

Adds SALESPERSON column. The column accepts variable length character data, and defaults to the current user.

ALTER TABLE CUSTOMERS DROP PHONE

The PHONE column is permanently removed from the table. Both the column definition and any data are deleted.

K 8**SQL Reference**

BEGIN TRANSACTION

The BEGIN TRANSACTION statement is used in both interactive and embedded SQL to activate transaction processing. Any subsequent SQL statements prior to a COMMIT or ROLLBACK statement constitute the transaction.

BEGIN TRANSACTION

Using BEGIN TRANSACTION

In interactive SQL, in order to use BEGIN TRANSACTION, the Auto-Transact prompt of the Global Environment window must be set to “Off”. If transaction processing is already active, an error message is displayed.

Examples

BEGIN TRANSACTION

Transaction processing is activated.

CLOSE (Embedded SQL)

The CLOSE statement is used only in embedded SQL to close a cursor.

CLOSE cursorvar

cursorvar

A variable assigned to a cursor in a previous DECLARE statement.

Examples

EXEC SQL CLOSE INVOICES_CURSOR ;

The cursor declared earlier in a DECLARE statement and opened in a previous OPEN statement is closed.

COMMIT

The COMMIT statement is used in both interactive and embedded SQL to execute a transaction. Any locks set by the transaction are released upon completion.

In embedded SQL, any open cursors are closed upon completion.

```
COMMIT {WORK}
```

WORK

The WORK keyword is used for clarity only.

Examples

```
ALTER TABLE CUSTOMERS MODIFY (ID INTEGER, DESCRIPTION VARCHAR  
NOT NULL)
```

```
INSERT INTO CUSTOMERS  
(SELECT * FROM POTENTIAL_CUSTOMERS)
```

```
COMMIT WORK
```

COMMIT ends the transaction consisting of the INSERT statement, writes the transaction to the COMMIT log, and signals the system to execute the transaction.

CREATE TABLE

The CREATE TABLE statement creates a new table of the specified column definitions. You can optionally assign the table to a specified Advanced Revelation volume.

```
CREATE TABLE table [column type {DEFAULT value} ⇨  
{column_def} ::] {ON vol}
```

table

A unique name for the table.

column

A unique name for a column. At least one column must be specified. Column specifications are enclosed in parentheses. Multiple column specifications are separated by commas.

type

One of the data type keywords for each *column* listed.

☞ See *Table K8-4*, “SQL Data Types” in this chapter for a listing.

DEFAULT

A keyword that assigns a default value to the column.

value

The value assigned may be:

- A literal value
- The NULL keyword
- The CURRENT keyword (DATETIME)
- The USER keyword (user name)

column_def

Column_def identifies additional definitions that may be applied to a column. More than one of the following keywords may be used (not delimited) in the order listed:

NOT NULL	Column must be assigned a non-null value.
UNIQUE	Used only with the NOT NULL keyword. <i>Column</i> may not contain any duplicate values.
{PRIMARY} KEY	Column is to be a primary key. At least one primary key column must be specified for <i>table</i> . Columns for a multipart key are allocated in the order specified. NOT NULL and UNIQUE requirements are implicit for PRIMARY KEY.
REFERENCES	Column is a foreign key in a specified list of <i>tables</i> . REFERENCES is followed by a <i>tables</i> parameter.
<i>tables</i>	A list of the table(s) to cross reference (delimited by commas).
REPEATING	Column is multivalued.

ON

A keyword to introduce the name of an existing Advanced Revelation volume.

vol

The name of an Advanced Revelation volume (defaults to current volume).

- ☛ For more information, see the chapter “Advanced Revelation Volumes” in *Filing Systems*.

Examples

```
CREATE TABLE CUSTOMERS (CUSTOMER_ID INTEGER KEY)
```

The CUSTOMERS table is created, with one column defined. The ID column requires integer data, and is the primary key.

```
CREATE TABLE PRODUCTS (PRODUCT_CODE INTEGER KEY, DESCRIPTION VARCHAR NOT NULL, PRICE DOLLARS NOT NULL)
```

The PRODUCTS table is created, with three columns defined. The PRODUCT_CODE column requires integer data, and is the primary key. The DESCRIPTION column allows variable length character data, and may not be null. The PRICE column requires monetary data and may not be null.

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY, CUSTOMER_ID INTEGER NOT NULL UNIQUE REFERENCES CUSTOMERS, CODE INTEGER NOT NULL UNIQUE REFERENCES PRODUCTS REPEATING, QUANTITY INTEGER DEFAULT 0 REPEATING)
```

The INVOICES table is created, with three columns defined. The INVOICE_NO column requires integer data, and is the primary key. The CUSTOMER_ID column requires integer data, must be a non-null, unique value, and is a foreign key for the CUSTOMERS table. The CODE column requires integer data, must be a non-null, unique value, is a foreign key for the PRODUCTS table, and accepts multiple values. The QUANTITY column requires integer data, defaults to 0 (zero), and accepts multiple values.

```
CREATE TABLE INVOICES (INVOICE_NO INTEGER KEY) ON DATAVOL
```

The INVOICES table is created on the DATAVOL volume with one column defined.

CREATE VIEW

The CREATE VIEW statement creates a view (a virtual table) using rows selected from one or more tables. You can optionally rename the columns for the view.

```
CREATE VIEW view {column ::} AS subquery {WITH CHECK OPTION}
```

view

The name of the view.

column

The name for the column or columns used in the view. Column identifiers are enclosed in parentheses. Multiple columns are separated by commas. The column identifiers may be the same or different from the column identifiers as used in the table(s).

AS

A keyword to identify the parameter following as a query.

query

A SELECT statement. Unions are not allowed.

WITH CHECK OPTION

A keyword to ensure that any update operations (UPDATE, INSERT, DELETE) on the base table for the view will conform to the definition of the view

Examples

```
CREATE VIEW CUSTOMER_CONTACT AS
  SELECT LAST_NAME, ADDRESS, PHONE FROM CUSTOMERS
```

The CUSTOMER_CONTACT view is created with three columns. The LAST_NAME, ADDRESS, and PHONE columns retain their names.

```
CREATE VIEW CUSTOMER_CONTACT (CUSTOMER, LOCATION, PHONE) AS
  SELECT LAST_NAME, ADDRESS, PHONE FROM CUSTOMERS
```

The CUSTOMER_CONTACT view is created as in the previous example. However, here the LAST_NAME and ADDRESS columns returned by the query are assigned the names CUSTOMER and LOCATION.

```
CREATE VIEW AZ_CUSTOMERS AS
  SELECT LAST_NAME, FIRST_NAME, CITY, PHONE FROM CUSTOMERS
    WHERE ST = "AZ"
  WITH CHECK OPTION
```

The AZ_CUSTOMERS view is created with four columns defined. The view is limited to those customers in Arizona. The WITH CHECK OPTION keyword ensures that data for customers outside of Arizona will not be accepted for the view.

DECLARE (Embedded SQL)

The DECLARE statement is used only in embedded SQL to provide a name and query expression for a cursor. DECLARE does not execute the query; a subsequent OPEN statement does this.

DECLARE variable CURSOR FOR query

variable

A variable used as the tag for a particular query definition. A maximum of eight cursors may be assigned.

CURSOR FOR

A keyword used to reference the query to be associated with the cursor *variable*.

query

Any valid SQL SELECT statement. You can substitute R/BASIC variables (preceded with a colon) for literal values in the SELECT statement.

Examples

```
EXEC SQL DECLARE CUSTOMERS_CURSOR CURSOR FOR
      SELECT CUSTOMER_ID, ST, ZIP FROM CUSTOMERS ORDER BY ZIP
END EXEC
* open cursor, execute selection conditions
EXEC SQL OPEN CUSTOMERS_CURSOR;
```

The CUSTOMERS_CURSOR is declared based on a query expression that returns the CUSTOMER_ID, ST and ZIP values from the CUSTOMERS table. The values are ordered from lowest to highest zip code. The query is then executed by the OPEN statement, and the resulting rows available in the cursor CUSTOMERS_CURSOR.

```
EXEC SQL
      DECLARE PRODUCTS_CURSOR CURSOR
      FOR SELECT PRODUCT_CODE, DESCRIPTION FROM PRODUCTS
      WHERE PRICE > 25.00
END EXEC
```

The cursor PRODUCTS_CURSOR is declared based on a query expression that returns the PRODUCT_CODE and DESCRIPTION values from the PRODUCTS table for products with a price greater than \$25.00.

DELETE

The **DELETE** statement removes rows from a table. You can include a **WHERE** clause to select rows to delete.

```
DELETE FROM table {WHERE search_conditions}
```

FROM

A keyword to identify the parameter following as a table.

table

The name of an existing table.

WHERE

A keyword to test row-by-row for deletion.

Warning! If you do not specify a **WHERE** clause, all rows in the table are deleted.

search_conditions

The *search_conditions* identify a **WHERE** clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a **WHERE** clause is:

```
WHERE expression comparison subquery {boolean} ::  
                                expression
```

- ☛ For a description of the parameters, see the **WHERE** keyword in “SQL Keywords”.

Examples

```
DELETE FROM CUSTOMERS WHERE CUSTOMER_ID = 330
```

*The row having a **CUSTOMER_ID** value of 330 is removed from the table.*

```
DELETE FROM PROJECTS
```

All rows are deleted from the table.

DELETE (Embedded SQL)

The DELETE statement is used in embedded SQL to remove rows from a table, optionally based on the current cursor position.

```
DELETE FROM table [ {WHERE search_conditions}
                  [ {WHERE CURRENT OF cursorvar} ]
```

FROM

A keyword to identify the parameter following as a table.

table

The name of an existing table.

WHERE

A keyword to test row-by-row for deletion. If unspecified, all rows are deleted.

search_conditions

The *search_conditions* identify a WHERE clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a WHERE clause is:

```
WHERE expression comparison subquery {boolean} ::
                                expression
```

- ☛ For a description of the parameters, see the WHERE keyword in “SQL Keywords”.

CURRENT OF

A keyword used to reference a cursor variable.

cursorvar

A variable assigned to a cursor in a previous DECLARE statement.

Examples

```
EXEC SQL DECLARE PRODUCTS_CURSOR CURSOR FOR
      SELECT * FROM PRODUCTS WITH STOCK_DATE < "01/01/1980"
END EXEC

EXEC SQL OPEN PRODUCTS_CURSOR ;

DONE = 0
LOOP
  RECORD = "" ;
```

```
EXEC SQL FETCH PRODUCTS_CURSOR INTO :RECORD;  
IF SQLSTATE = 100 THEN DONE = 1  
UNTIL DONE  
EXEC SQL DELETE FROM PRODUCTS  
WHERE CURRENT OF PRODUCTS_CURSOR  
END EXEC  
REPEAT
```

```
EXEC SQL CLOSE PRODUCTS_CURSOR ;  
STOP
```

The PRODUCTS_CURSOR is declared and opened using a query expression selecting all rows in which the original stock date is earlier than 1980. The FETCH statement is used to position the cursor on the appropriate row, and the DELETE FROM statement then deletes the current row based on that position in PRODUCTS_CURSOR.

K 8

SQL Reference

DROP TABLE

The DROP TABLE statement removes a table.

- ☞ To use DROP TABLE in compiled or embedded SQL, use the EXEC_SQL sub-routine.

DROP TABLE table

table

The name of the table to delete.

Examples

```
DROP TABLE PRODUCTS
```

The PRODUCTS table is removed from the database.

DROP VIEW

The DROP VIEW statement removes a view definition.

- ☞ To use DROP VIEW in compiled or embedded SQL, use the EXEC_SQL sub-routine.

DROP VIEW view

view

The name of the view to delete.

Examples

```
DROP VIEW CUSTOMER_CONTACT
```

The CUSTOMER_CONTACT view definition is removed from the database.

FETCH (Embedded SQL)

The FETCH statement is used only in embedded SQL to position a cursor and assign the referenced row to a target parameter (R/BASIC variable).

```
FETCH {position} FROM cursorvar INTO [target_parameter ::]
```

position

Position sets the position of the cursor referenced by *cursorvar*. If *position* is not specified, the default value is the next row forward. The *position* value may be any one of the following keywords:

Keyword	Meaning
NEXT	The next row forward (the default <i>position</i> value)
PRIOR	The next row back
FIRST	The first absolute row in <i>cursorvar</i>
LAST	The last absolute row in <i>cursorvar</i>
ABSOLUTE <i>n</i>	The <i>n</i> th absolute row in <i>cursorvar</i> . <i>N</i> may be a non-zero positive or negative integer. If negative, the position is <i>n</i> rows back from the end.
RELATIVE <i>n</i>	The <i>n</i> th row in <i>cursorvar</i> based on the current cursor position. <i>N</i> may be a non-zero positive or negative integer.

FROM

A keyword indicating that the following identifier is a cursor.

cursorvar

A variable assigned to a cursor in a previous DECLARE statement.

INTO

A keyword used with FETCH to identify the parameter following as a local variable to store the returned row.

target_parameter

One or more R/BASIC variables to store the column value(s) in the returned row. Each *target_parameter* specified must be preceded by a colon.

If only one *target_parameter* is specified, all column values from the *FETCH* statement are returned as a dynamic array. If multiple *target_parameters* are specified, each column value is returned into an individual parameter in order. If there are fewer *target_parameters* than column values, extra column values are returned as a dynamic array in the last *target_parameter*.

SQLSTATE Values

The values returned in the *SQLSTATE* system variable are:

Value	Meaning
0	Row returned successfully
100	Cursor done: last row fetched (<i>target_parameter</i> is null)
< 0	Error

Examples

```
CITY = "" ; ST = "" ; ZIP = ""
EXEC SQL FETCH CUSTOMER_CURSOR INTO :CITY,:ST,:ZIP ;
```

The FETCH statement returns columns from the current row into the target parameters CITY, ST, and ZIP. If more than three columns are returned, the target parameter ZIP is returned as a dynamic array with one field per column.

```
DONE = 0
LOOP
  * fetch contents of next row into variable RECORD
  RECORD = ""
  EXEC SQL FETCH CUSTOMERS_CURSOR INTO :RECORD ;
  BEGIN CASE
    CASE SQLSTATE = 0
      * normal return, more rows available
    CASE SQLSTATE = 100
      * no more rows available
      DONE = 1
    CASE 1
      MSG("Error %1% during FETCH","", "",SQLSTATE)
      DONE = 1
  END CASE
UNTIL DONE REPEAT
```

In this program fragment, the FETCH statement is used to return the current row from the cursor indicated by CUSTOMERS_CURSOR (a DECLARE and OPEN statement must precede this example). The columns are returned as a dynamic array into the target parameter RECORD. The CASE statement following the FETCH statement tests whether more rows remain, the cursor is exhausted, or there has been an error.

```

DONE = 0 ; POS = 1
LOOP
  * fetch contents of next row into variable RECORD
  RECORD = ""
  EXEC SQL FETCH RELATIVE POS CUSTOMERS_CURSOR INTO :RECORD ;
  GOSUB DISPLAY_RECORD: ; * subroutine not displayed here
  RESP = "F"
  CALL MSG("[F]orward, [B]ackward, [Q]uit?", "RC", RESP, "")
  BEGIN CASE
    CASE RESP = "Q" ; DONE = 1
    CASE RESP = "B" ; IF POS > 0 THEN POS = -1
    CASE RESP = "F" ; IF POS < 0 THEN POS = 1
    CASE 1
  END CASE
UNTIL DONE REPEAT
STOP

```

*This example illustrates the use of a scrolled cursor. The **FETCH** statement contains an explicit positioning parameter (**RELATIVE POS**) that positions the cursor relative to the current row. The default is to move one row forward. If the user responds with "B" the position of the cursor moves backward toward the first row in the cursor. Subroutines referenced in this program fragment are not shown here.*

INSERT

The **INSERT** statement adds new rows to an existing table. You can assign literal values or column values from a query to specified columns.

The **INSERT** statement format is the same in both interactive and embedded SQL. However, literal values can be passed as parameters in embedded SQL.

```

INSERT INTO table [columns ::] [VALUES [value ::]
                                     query

```

INTO

A keyword used with **INSERT** to identify the identifier following as the table to store the new row.

table

The name of an existing table.

columns

The name of an existing column or columns within *table*. The columns are enclosed in parentheses. Multiple columns are separated by commas.

Any columns not assigned a value by a **VALUES** clause or query default to null, unless prohibited by the column definition. An attempt to assign a null value to a column defined with **NOT NULL** will generate an error.

VALUES

A keyword indicating the parameter following is a value or values. The values are assigned to the columns in corresponding order.

If column assignments are not specified, the key column(s) is first assigned (based on the order in which they were specified for multipart keys), then the remaining columns are assigned according to their defined position in the dictionary for the table.

value

A literal value(s). The value(s) is enclosed in parentheses. Multiple values are separated by commas.

query

A restricted **SELECT** statement (no **ORDER BY**, **UNION**, or options). Multiple rows may be inserted with the query. The column values must conform to the established column definitions in *table*.

If column assignments are not specified, the key column(s) is first assigned (based on the order in which they were specified for multipart keys), then the remaining columns are assigned according to their defined position in the dictionary for the table.

The referenced table may be copied to *table* by selecting all rows in the table.

- ☛ A row may not be inserted into the same table (the *query* may not be applied to *table*.)

Examples

```
INSERT INTO CUSTOMERS (CUSTOMER_ID, LAST_NAME) VALUES
(311, "Wallace")
```

A new row having a CUSTOMER_ID of 311 and a LAST_NAME of "Wallace" is added to the CUSTOMERS table.

```
INSERT INTO CUSTOMERS (CUSTOMER_ID, LAST NAME, FIRST NAME)
SELECT CUSTOMER_ID, LAST NAME, FIRST NAME FROM CUSTOMERS_88
WHERE CUSTOMERS_88.CUSTOMER_ID = 311
```

Information from a row from last year's table of customers is inserted in the the current CUSTOMERS table. The row is identified by the primary key CUSTOMER_ID.

```
INSERT INTO OLD_PRODUCTS
SELECT * FROM PRODUCTS
```

All rows in the PRODUCTS table are copied to the OLD_PRODUCTS table. The tables must have identical structures for this operation to work properly.

OPEN (Embedded SQL)

The OPEN statement is used only in embedded SQL to activate a cursor. The cursor is opened by executing the query associated with the cursor at declaration (the DECLARE statement).

OPEN cursorvar

cursorvar

A variable assigned to a cursor in a previous DECLARE statement

Examples

```
EXEC SQL DECLARE CUSTOMERS_CURSOR CURSOR FOR
      SELECT CUSTOMER_ID, ST FROM CUSTOMERS ORDER BY ZIP
END EXEC
* open cursor, execute selection conditions
EXEC SQL OPEN CUSTOMERS_CURSOR;
```

The cursor CUSTOMERS_CURSOR is declared based on the query expression. The OPEN statement then executes the query expression and loads a cursor with the resulting rows. Subsequent embedded SQL statements can access the cursor using the cursor name CUSTOMERS_CURSOR.

ROLLBACK

The ROLLBACK statement is used in both interactive and embedded SQL to terminate the execution of a transaction. ROLLBACK guarantees that none of the operations in the transaction are committed to the database. Any locks set by the transaction are released upon completion.

In embedded SQL, any open cursors are closed upon completion.

ROLLBACK {WORK}

WORK

The WORK keyword is used for clarity only.

Examples

```
INSERT INTO OLD_PRODUCTS
  SELECT * FROM PRODUCTS
```

```
ROLLBACK WORK
```

The ROLLBACK operation aborts the transaction, removing it from the buffer (transaction file).

SELECT

The SELECT statement constructs queries. You can specify search conditions, nest and combine queries, and group and order the selected rows.

```
SELECT {function} [{DISTINCT} [column ::]] FROM [table ::] ⇨
                                     {ALL}

{WHERE search_conditions} ⇨
{GROUP BY columns} {HAVING search_conditions} ⇨
{ORDER BY columns order} ⇨
{UNION {ALL} query ::} ⇨
{options}
```

function

One of the function keywords: AVG, SUM, COUNT, MAX or MIN. The argument to the functions is enclosed in parentheses, and immediately follows the function. For example,

```
AVG(column)
```

```
MAX(column1 + column2)
```

Unless the DISTINCT keyword is specified, the function keywords are applied to *all* rows (including duplicates).

DISTINCT

Specifies that the query return only one representative row, among any duplicate rows selected.

✎ DISTINCT may not be used with the function keyword COUNT(*).

ALL

The default mode. ALL specifies that any duplicate rows selected will be returned.

column

A column, a list of columns (delimited by commas), a column equation, or * (asterisk).

An example of a column equation might be:

```
SELECT (field1 + field2) FROM table
```

The * (asterisk) operator indicates all columns. When used without a *table* parameter, all columns in all tables are selected.

- ☛ Use the full column name (*table.column*) to avoid ambiguity when referencing more than one table.

FROM

A keyword to identify the parameter following as a table or tables

table

The name of an existing table(s) or view(s). If multiple tables or views are specified, they are separated by commas.

WHERE

A keyword to test row-by-row for selection.

search_conditions

The *search_conditions* identify a WHERE clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a WHERE clause is:

```
WHERE expression comparison subquery {boolean} ::  
      expression
```

- ☛ For a description of the parameters, see the WHERE keyword in “SQL Keywords”.

GROUP BY

A keyword that collects the selected rows into groups based on the *columns* specified.

columns

The name of an existing column or columns (separated by commas)

HAVING

A keyword to test groups of rows as a whole: the entire group must meet the conditions. When it is not preceded by a GROUP BY clause, a HAVING operation will test the complete table.

search_conditions

The *search_conditions* identify a HAVING clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The WHERE and HAVING keywords use the same syntax:

```
HAVING expression comparison subquery {boolean} ::  
                                expression
```

- ☛ For a description of the parameters, see the WHERE keyword in the “SQL Keywords”.

ORDER BY

A keyword to sort the selected rows according to the *columns* and *order* specified.

columns

The name of an existing column(s) or the column position(s) as defined in the dictionary. Multiple *columns* are separated by commas.

order

One of the order keywords ASC or DESC to indicate an ascending (lowest ASCII value first) or descending (highest first) sort. The *order* defaults to ASC.

UNION

A keyword that combines the results of two queries to produce a single table. Duplicate rows are eliminated, unless ALL is specified. Associated UNION clauses are enclosed in parentheses.

ALL

When used with UNION, the ALL keyword includes duplicate rows in the results table returned by the UNION operation.

query

A SELECT statement.

options

The *options* are Advanced Revelation extensions. Two keyword *options* are available:

Option	Meaning
SUPPRESS DISPLAY	The results table is not displayed.
TO PRINTER	The results table is sent to the default printer.

Examples

```
SELECT * FROM CUSTOMERS
```

All of the columns of the CUSTOMERS table are selected.

```
SELECT AVG(PRICE) FROM INVOICES
```

The average value of the PRICE column for all rows is returned.

```
SELECT CUSTOMER_ID, (QUANTITY * PRICE) FROM INVOICES, PRODUCTS
WHERE CODE = PRODUCT_CODE
```

The subtotal is returned for each customer order.

```
SELECT LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST, ZIP FROM
CUSTOMERS
```

The values for the LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST and ZIP columns for all rows are returned.

```
SELECT LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST, ZIP FROM
CUSTOMERS, INVOICES
WHERE INVOICES.DATE = "22 October 1989"
```

The values for the LAST_NAME, FIRST_NAME, ADDRESS, CITY, ST and ZIP columns for any rows in the INVOICES table having a DATE value of "22 October 1989" are returned.

```
SELECT DISTINCT DESCRIPTION, QUANTITY FROM PRODUCTS, INVOICES
WHERE PRODUCT_CODE = CODE
ORDER BY QUANTITY
```

The values for the PRODUCT_CODE and QUANTITY columns of the PRODUCTS and INVOICES tables are selected for any rows having identical codes. The DISTINCT keyword specifies that each product will be selected only once, no matter how many entries there might be. The ORDER BY keyword sorts the result table from least to greatest QUANTITY values.


```
SELECT DESCRIPTION, PRICE FROM PRODUCTS
GROUP BY PRICE
HAVING AVG(PRICE) > 40.00
ORDER BY PRICE DESC
```

The query selects only those PRICE groups that have an average price greater than \$40.00. The products within each group selected are sorted from highest to lowest (DESC) price.

```
SELECT DESCRIPTION, PRICE, QUANTITY FROM PRODUCTS ,INVOICES
WHERE SUM(QUANTITY) >= 2
GROUP BY PRICE
HAVING AVG(PRICE) > 40.00
ORDER BY PRICE DESC
```

This query differs from the previous example in that the GROUP BY, HAVING and ORDER BY operations are applied only to those products that have two or more orders.

```
SELECT FIRST_NAME, LAST_NAME, ADDRESS, CITY, ST, ZIP FROM
CUSTOMERS
WHERE CUSTOMER_ID = ANY
(SELECT CUSTOMER_ID FROM CUSTOMERS_88)
```

The customer mailing information from the current table is selected only for previous customers.

```
SELECT INVOICES.CODE, INVOICES_88.CODE, SUM(INVOICES.QUANTITY *
PRODUCTS.PRICE), SUM(INVOICES_88.QUANTITY * PRODUCTS_88.PRICE)
FROM INVOICES, INVOICES_88, PRODUCTS, PRODUCTS_88
WHERE INVOICES.CODE =
(SELECT CODE FROM INVOICES_88)
```

A join table of the retail value of products sold for this year and last is returned.

SELECT (Embedded SQL)

The SELECT statement is used in embedded SQL to select one or more rows and assign the row(s) to a target parameter (R/BASIC variable).

In Advanced Revelation SQL, the embedded SELECT statement can return multiple rows. This is an extension to the standard SQL embedded SELECT that returns only single rows.

```
SELECT [DISTINCT] [column ::] INTO target_parameter ⇐
[ALL]

{flag} FROM [table ::] {WHERE search_conditions} ⇐
{GROUP BY columns} {HAVING search_conditions} ⇐
{ORDER BY columns order} {options}
```

DISTINCT

Specifies that the query return only one representative row, among any duplicate rows selected.

- ✎ DISTINCT may not be used with the function keyword COUNT(*).

ALL

The default mode. ALL specifies that any duplicate rows selected will be returned.

column

A column, a list of columns (delimited by commas), a column equation, or * (asterisk).

An example of an column equation might be:

```
SELECT (field1 + field2) FROM table
```

The * (asterisk) operator indicates all rows. When used without a *table* parameter, all rows in all tables are selected.

- ✎ Use the full column name (*table.column*) when referencing more than one table.

INTO

A keyword used to identify the parameter following as a local variable(s) to store the new row.

target_parameter

One or more R/BASIC variables to store the column value(s) in the returned row. Each *target_parameter* specified must be preceded by a colon.

If only one *target_parameter* is specified, all column values from the SELECT statement are returned as a dynamic array. If multiple *target_parameters* are specified, each column value is returned into an individual parameter in order. If there are fewer *target_parameters* than column values, extra column values are returned as a dynamic array in the last *target_parameter*.

If more than one row is returned by the SELECT statement, each row in *target_parameter* is delimited with @RM.

flag

An R/BASIC variable used to test for a null assignment to *column* or *target_parameter*. If the *column* or *target_parameter* is null, *flag* is assigned a value of 1, otherwise *flag* is assigned a value of zero.

FROM

A keyword to identify the parameter following as a table or tables

table

The name of an existing table(s) or view(s). If multiple tables or views are specified, they are separated by commas.

WHERE

A keyword to test row-by-row for selection.

search_conditions

The *search_conditions* identify a WHERE clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a WHERE clause is:

```
WHERE expression comparison subquery {boolean} ::  
                                expression
```

- ☛ For a description of the parameters, see the WHERE keyword in “SQL Keywords”.

GROUP BY

A keyword that collects the selected rows into groups based on the *columns* specified.

columns

The name of an existing column or columns (separated by commas).

HAVING

A keyword to test groups of rows as a whole: the entire group must meet the conditions. When it is not preceded by a GROUP BY clause, a HAVING operation will test the complete table.

search_conditions

The *search_conditions* identify a HAVING clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The WHERE and HAVING keywords use the same syntax:

```
HAVING expression comparison subquery {boolean} ::  
                                expression
```

- ☛ For a description of the parameters, see the WHERE keyword in the “SQL Keywords”.

ORDER BY

A keyword to sort the selected rows according to the *columns* and *order* specified.

columns

The column position(s) as defined in the dictionary. Do not use a column identifier. Multiple *columns* are separated by commas.

order

One of the order keywords ASC or DESC to indicate an ascending (lowest ASCII value first) or descending (highest first) sort. The *order* defaults to ASC.

options

The *options* are Advanced Revelation extensions. Two keyword *options* are available:

Option	Meaning
SUPPRESS DISPLAY	The results table is not displayed.
TO PRINTER	The results table is sent to the default printer.

SQLSTATE Values

The values returned in the SQLSTATE system variable are:

Value	Meaning
0	Successful operation. One row returned.
100	No rows returned (or no more rows to return).
101	All rows returned.
102	Multiple rows returned (not all).
< 0	Error condition.

Examples

```

DONE = 0
LOOP
  RECORD = ""
  EXEC SQL
    SELECT FIRST NAME, LAST NAME INTO :RECORD
    FROM CUSTOMERS WHERE ST = "CA"
  END EXEC
  IF SQLSTATE = 102 THEN
    * more rows to be returned
    GOSUB EXTRACT.ROWS
  END ELSE
    * all (possible) rows returned
    DONE = 1
  END
END

```

```

UNTIL DONE REPEAT
GOSUB EXTRACT.ROWS:
STOP
EXTRACT.ROWS:
ROW.CNT = COUNT( RECORD, @RM ) + 1
FOR ROW.CTR = 1 TO ROW.CNT
    NEXT.ROW = FIELD( RECORD, @RM, ROW.CTR )
    GOSUB PROCESS ; * subroutine not shown
NEXT ROW.CTR
RETURN

```

The program uses an embedded SQL SELECT statement to return all rows in the CUSTOMERS tables in which the value in the ST column is "CA". Up to 64K of rows are returned in the target parameter RECORD. If more rows remain, SQLSTATE is set to 102, the program processes the current rows, and then loops and fetches further rows. Rows are extracted from the target parameter RECORD using an R/BASIC FIELD statement. Individual rows in the target parameter are delimited with record marks (@RM).

SET TRANSACTION

The SET TRANSACTION statement applies the specified concurrency controls to any subsequent transactions. If transaction processing is not enabled, none of the settings are applied.

- ☛ A SET TRANSACTION statement may only be used after you have issued a BEGIN TRANSACTION statement and before your first data manipulation statement (INSERT, UPDATE, DELETE, SELECT, FETCH, ALTER TABLE).

The SET TRANSACTION settings will override any relevant environmental settings. These new settings are changed by executing another SET TRANSACTION statement or by resetting or logging off the system.

```

SET TRANSACTION CONSISTENCY LEVEL level { [ READ ONLY ]
                                           [ READ WRITE ] }

```

CONSISTENCY LEVEL

A keyword identifying the type and duration of locks for implicit locking. The *level* defaults to the environmental setting for consistency level (see the chapter "Managing the System").

level

A number specifying one of the available level settings:

Level	Meaning
0	Disables implicit locking.
1	Enables exclusive locking during updates. The lock is held for the duration of the update.
2	Enables exclusive locking during updates. The lock is held for the duration of each transaction.
3	Enables exclusive locking during updates. The lock is held for the duration of each transaction. In addition, shared locking is enabled during queries. This lock is held for the duration of the query.
4	Enables exclusive table locking during updates and shared table locking during queries. The locks are held for the duration of each transaction. The <i>Implicit Locking</i> setting at the Global Environment window must be set to File locking in order to use level 4. This level guarantees serializable transactions when <i>Commit Protection</i> is set at the Global Environment window.

- ☞ For more information, see the “Locking” chapter in *Networks*, and the **CONSISTENCY LEVEL** keyword in “SQL Keywords”.

READ ONLY**READ WRITE**

The **READ ONLY** and **READ WRITE** keywords are included for compatibility only. Currently, all transactions are read/write.

Examples

```
SET TRANSACTION CONSISTENCY LEVEL 3
```

Exclusive locking is enabled during updates for the duration of each transaction, and shared locking is enabled during queries for the duration of the query.

UPDATE

The UPDATE statement assigns new column values to existing rows.

```
UPDATE table SET [column = value ::] {WHERE search_conditions}
```

table

The name of an existing table or view.

SET

A keyword that introduces a column assignment(s).

column

The name of an existing column in *table*.

value

The value to assign to *column*. *Value* may be:

- A literal value
- The name of a column
- A column equation
- A function
- The NULL keyword
- The USER keyword

If more than one assignment clause (*column = value*) is specified, the clauses are separated by commas.

WHERE

A keyword to test row-by-row for processing.

Warning! If a WHERE clause is not specified, all rows in the table will be updated.

search_conditions

The *search_conditions* identify a WHERE clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a WHERE clause is:

```
WHERE expression comparison subquery {boolean} ::  
expression
```

- ☛ For a description of the parameters, see the WHERE keyword in “SQL Keywords”.

Examples

```
UPDATE INVOICES SET QUANTITY = 0
```

The QUANTITY column value in every row is assigned the value 0.

```
UPDATE CUSTOMERS SET LAST_NAME = "Rose"
WHERE FIRST_NAME = "Danika"
```

The row having a LAST_NAME value of "Rose" is assigned "Danika" as the value of the FIRST_NAME column.

UPDATE (Embedded SQL)

The UPDATE statement is used in embedded SQL to assign new column values to rows, optionally based on the current cursor position.

```
UPDATE table SET [column = value {flag} ::] ⇨
[ {WHERE search_conditions} ]
[ {WHERE CURRENT OF cursorvar} ]
```

table

The name of an existing table or view.

SET

A keyword that introduces a column assignment(s).

column

The name of an existing column in *table*.

value

The value to assign to *column*. *Value* may be:

- A literal value
- The name of a column
- A column equation
- A target parameter (R/BASIC variable)
- The NULL keyword
- The USER keyword

If more than one assignment clause (*column = value*) is specified, the clauses are separated by commas.

flag

A local variable used to test for a null assignment to *column* or *target_parameter*. If the *column* or *target_parameter* is null, *flag* is assigned a value of -1, otherwise *flag* is assigned a value of zero.

WHERE

A keyword to test row-by-row for updating.

Warning! If a WHERE clause is not specified, all rows in the table will be updated.

search_conditions

The *search_conditions* identify a WHERE clause. The clause can be a test for the existence of data, or a condition(s) to be met.

The syntax for a WHERE clause is:

```
WHERE expression comparison subquery {boolean} ::
      expression
```

- ☛ For a description of the parameters, see the WHERE keyword in "SQL Keywords".

CURRENT OF

A keyword used to reference a cursor variable

cursorvar

A variable assigned to a cursor in a previous DECLARE statement. The cursor must be updateable.

Examples

```
DECLARE SUBROUTINE MSG
EXEC SQL WHENEVER NOT FOUND GOSUB DONE ;

* initialize cursor name and conditions
EXEC SQL DECLARE PRODUCTS_CURSOR CURSOR FOR
      SELECT * FROM PRODUCTS
END EXEC

* open cursor, execute selection conditions
EXEC SQL OPEN PRODUCTS_CURSOR ;
DONE = 0
LOOP
  * fetch contents of next row into variable RECORD
  RECORD = ""
```

```
EXEC SQL FETCH PRODUCTS_CURSOR INTO :RECORD ;
UNTIL DONE
  KEY = RECORD<1>
  NEW_VALUE = ""
  MSG("Enter a new price for row ":KEY,"R",NEW_VALUE,"")
  IF NEW_VALUE THEN
    * update table via cursor update
    EXEC SQL UPDATE PRODUCTS
      SET PRICE = :NEW_VALUE
      WHERE CURRENT OF PRODUCTS_CURSOR
    END EXEC
  END
REPEAT
STOP

DONE :
DONE = 1
RETURN
```

The cursor PRODUCTS_CURSOR is declared to return all columns from the PRODUCTS table. The cursor is then executed by the OPEN statement and each row selected is successively fetched and displayed, prompting the user whether to update the PRICE value. If the user enters a new value for the PRICE column, the UPDATE statement is used to update the table based on the current cursor position.

Chapter L 1

Introduction to Networks

Advanced Revelation is an ideal environment for use on Local Area Networks (LANs). Whether you are designing new applications or simply using existing ones, you will find that Advanced Revelation's efficient design and powerful tools help your network become the productivity tool it was meant to be.

In this chapter, you will find general information about how to use Advanced Revelation with your network. Subsequent chapters in this section describe how to manage your Advanced Revelation system on a network and provide details about such important issues as file and record locking.

Introduction	L 1 . 1
Advanced Revelation and Networks	L 1 . 3



Introduction

If you work on a Local Area Network (LAN) — or if you plan to do so in the future — you will find that Advanced Revelation is an ideal application development environment. Advanced Revelation's powerful application development tools and flexible architecture allow you to build applications easily while still taking full advantage of the capabilities of the network.

In using Advanced Revelation on a network, you will enjoy these benefits:

- Advanced Revelation's efficient filing system minimizes network "traffic". Advanced Revelation is particularly efficient for high-volume, transaction-based applications.
- Response over the network remains the same even as files grow larger.
- You can store Advanced Revelation data across any combination of local and network media, including multiple servers and remote devices.
- Advanced Revelation offers a full array of options for coordinating data access with other users ("locking"). You can lock files or records, and you can have Advanced Revelation lock automatically or take full control of over how and when locks are set.
- You can even use your network for distributed processing. Advanced Revelation's indexing system allows you to share the task of indexing across multiple workstations. In addition, you can use database servers for optimizing data access, especially for large-volume processes and reports.

About Networks

A network is a group of computers or "workstations" that are linked together to share resources and data. Typically, there are two resources that network workstations share: disk storage and printers. More sophisticated networks may also offer the opportunity to share a resource such as a communication link to another network or larger computer.

Networks can be simple or complex. A simple network, for instance, might link two PCs together using a length of wire similar to that used for cable television. Large networks, or WANs (Wide Area Networks), on the other hand, might actually span huge areas from buildings to entire states, linking workstations through cabling, telephones, infrared and microwave channels, and more.

Advantages of a Network

The first benefit of a network is cost. By sharing resources, you save the cost of equipping each PC in your company with its own hard disk and printer. Instead, each user on the network can share disks and printers, treating them as if they were local to the station.

A far more important benefit than sharing *hardware*, though, is sharing *information*. By tying in to a network, users can access a single, central database of information.

For example, you can maintain a central database of customers and invoices on a network. By having each user on the network use that central database, you avoid having separate copies of the data for each user. Not only does having a central database save time and space, but by maintaining just one copy of the data, you ensure the integrity of the database. With only one copy, you do not need to worry about which copy of the data is current.

Types of Networks

There are many types of networks available today. Features and functions vary from one brand to the next, but the basic function of all networks is the same.

Because of the way Advanced Revelation communicates with your network, you rarely have to concern yourself with network differences — in fact, it is only when you configure Advanced Revelation for your network. Advanced Revelation is designed to minimize the need for user interaction with the network.

Network Hardware Network hardware includes everything required to physically link computers together. This usually consists of a card that is inserted into a PC, plus cabling to connect it to other PCs and to the central PC.

Your choice of network hardware affects the cost and performance of your network. However, as a rule, the hardware has little to do with how you run software such as Advanced Revelation on your network.

Network Software Far more important to your work with Advanced Revelation is the network software that you use. The network software, or network operating system (NOS), does for your network what your PC operating system (DOS or OS/2) does for your PC: it coordinates activity on the network and provides uniform, high-level access to the hardware.

For example, your network software will determine how you can access printers and hard disks on your network. The network operating system also determines such factors in your network as security and data coordination (locking).

☞ Whenever you see reference in Advanced Revelation to a “network type” or simply “network”, it *always* means network *software*. For example, if you see reference to an “IBM” or “Novell” network, this refers to the brand of network operating system software that you are using. Do not confuse network hardware and software!

Servers and Workstations

Most networks make a distinction between workstations (usually PCs) and servers. A workstation is the computer that you use for your own processing. For instance, when you use Advanced Revelation on a network, you are actually executing Advanced Revelation commands and processes at your local station (even though they are stored on the network).

When tied into a network, workstations use the resources of a “server”. The server is a computer dedicated to “serving” requests from workstations. Most requests take the form of a request for data (using the server’s disks) or for printing (using a printer attached to the server).

The type of server just described is a “file server”. A file server accepts requests for files and usually provides a way for network users to print to a central printer. The file server does little or no processing of the data — any processing tasks are left to the workstation itself.

A different type of server is a “database server”. This type of server is specially designed to accept not just requests for files, but requests for specific records. For example, a database server will allow you to send a request such as this: “please retrieve all records for customers with open invoices, sorted according to zip code”. The database server actually examines the data and returns only the records that meet your request.

Advanced Revelation and Networks

In many ways, using Advanced Revelation on a network is no different from using it on a stand-alone PC. Certainly much of what you do with Advanced Revelation is equally applicable in a single-user or a network environment: building and using windows, maintaining your database, running reports, and so forth.

However, there are important differences. For one thing, you must “bump” your single-user copy of Advanced Revelation before you can use it on a network. In addition, you must coordinate data access with other users (set “locks”). The rest of this chapter addresses general issues that arise when you use Advanced Revelation on a network. For additional information on using Advanced Revelation on a network, see the chapters “Managing Advanced Revelation on a Network” and “Locking”, which follow.

Network Compatibility

You can use Advanced Revelation on all major networks. There are network “drivers” available for the best-known networks (see “Preparing Advanced Revelation for a Network” below).

If you use a network that does not appear on the list of supported networks, you will probably still be able to use Advanced Revelation. Even lesser-known networks generally support or emulate one of the networks supported by Advanced

Revelation. In addition, Advanced Revelation supports a “generic” network that uses the very popular Microsoft MS-DOS 3.1 protocol.

For example, a small network may use the protocol of a better-known system such as IBM or Novell. As long as the emulation is complete, Advanced Revelation will work fine because it can issue commands as if communicating with the better-known system.

- ☞ As a rule, Advanced Revelation requires that you use a “dedicated” file server. This means that you cannot use the server as both a server *and* a workstation. Even if your network allows the use of non-dedicated servers, you may not be able to use Advanced Revelation in this configuration. Consult your network documentation and the Network Bump Disk documentation (see below) for details on using non-dedicated servers.

Preparing Advanced Revelation for a Network (“Bumping”)

When you buy a copy of Advanced Revelation, you buy a single-user system. To use Advanced Revelation on a network, you must “bump” it with one or more Network Bump Disks. Bump disks are sold separately.

- ☞ *Never* attempt to share data with other users on a network unless you have bumped your copy of Advanced Revelation. If you do not bump first, data corruption is inevitable.

Bumping a copy of Advanced Revelation accomplishes two tasks:

- It allows you to support multiple users.
- It allows you to choose the type of network that you are running on.

The first time you bump your copy of Advanced Revelation, it turns from a single-user system into a system that supports up to four users at a time. If more than four users need to use Advanced Revelation concurrently, you can bump again with a new bump disk. Each bump adds four more users onto your copy of Advanced Revelation. For example, to support a total of 16 users, you need to bump four times using four bump disks.

When you bump, you also specify the type of network that you are running on. This enables Advanced Revelation to install the appropriate network driver for your system. Although this is part of the bump process, you can change network drivers at any time, should you ever need to do so.

- ☞ Instructions on bumping your system and choosing a network driver are provided with the Network Bump Disk.

Coordinating Access to Data

Because multiple users on the network have access to the same data, more than one person might attempt to update a record at the same time. This access conflict could result in either:

- one user overwriting another's changes to a record without either one knowing, or
- the file being corrupted and data lost.

To avoid conflicts of this sort, Advanced Revelation allows you to "lock" records and files as you use them. By locking a file or record, you are saying "this file (or record) is in use". Other users who attempt to lock the same file or record are not allowed to do so.

You can update a file or record while you have it locked. Since you are the only user who has the file or record locked, you are assured that no one else will be updating it at the same time. When you are through updating, you unlock the file or record. Other users can then lock and update the file or record.

Advanced Revelation offers you different levels of locking files and records. You can lock an entire file or just an individual record. In addition, you can choose different degrees of locking, including "exclusive", "shared", and "coordinated" locks.

- ☛ Information about locking and lock types appears in the chapter "Locking" in this section.

Writing Network Applications

Advanced Revelation allows you to write applications that can be used in both single-user and network environments. For example, if you are working on a single PC, but anticipate that your application will be run on a network, you can build all locking into your application as you develop it.

Most tools and utilities that you use in Advanced Revelation are network-ready. For example, the Advanced Revelation editor does record locking as you edit records. Advanced Revelation windows even allow you to specify the type of locking that you wish to use by simply entering your choice at a prompt in Paint. If you use R/BASIC, Advanced Revelation's programming language, you take complete control over how and when locking is done.

If you run in a single-user environment, locks are ignored. However, when you move to a network, the locks built into your application are passed through to the network, assuring you of coordination with other users.

Network Users

Advanced Revelation makes a distinction between workstations and users. Advanced Revelation considers anyone to be a “user” who logs into Advanced Revelation over a network. This is different from the number of workstations physically attached to the network.

For example, in Advanced Revelation terminology “four users” means that there are four users logged into Advanced Revelation simultaneously. These four users can be logged onto any four workstations on the network.

- ☛ If you are working with OS/2, you are still considered only one user, even though you can run multiple sessions from your workstation. For details on other multi-session systems, see the documentation that accompanies your network software and the Advanced Revelation optional module for that system.

Station Number

Each network workstation has a unique identification for Advanced Revelation. This “station number” is usually derived directly from the network, using a logical or physical address. The station number is important in a network environment because it is used to uniquely identify processes, files, and records.

- ☛ You should be sure that every workstation on your network can return a unique identification.

If you are an R/BASIC programmer, the station number is available to you in the system variable @STATION.

Network Information Report

If you need to see information about your Advanced Revelation network configuration, use the WHO command at TCL. The WHO window displays the number of network users to which you have bumped, the type of network driver that you have installed, and your station number.

Chapter L 2

Managing Advanced Revelation on a Network

When you use Advanced Revelation on a network, you will see very little difference from using it on a stand-alone PC. However, there are some issues you need to be aware of when creating, maintaining, or using network applications.

This chapter addresses those issues. Here you will find information about everything from file types to transaction processing, and how these are used in a network version of Advanced Revelation.

Managing Files on a Network	L 2 . 1
The Advanced Revelation Environment	L 2 . 2
Executing Network Commands	L 2 . 5
Managing Indexing on a Network	L 2 . 6
Other Network Management Issues	L 2 . 8



Managing Files on a Network

Under most circumstances, using files on a network is no different than using files on a stand-alone system. However, because you will be sharing data with other users, certain issues arise that do not apply to a single-user system.

Accessing Files

When using a network, you can access any combination of local and network files concurrently. For example, you may be accessing shared customer and invoice files on the network, while maintaining a local file for your own to-do list or contact information.

If you do access local files, remember that others on the network will probably not be able to read or update these files. Therefore be careful about storing data locally if you wish others to have access to it.

Although you can attach any combination of network and local files, you should only access network files if you are logged onto a network version of Advanced Revelation. See “Logging on to Advanced Revelation” elsewhere in this chapter for details.

To access files on a network, simply attach them as you would a local file. You can use the Attach window or the TCL command ATTACH, or you can create a volume that includes the network location of your files.

- For information on attaching files and creating volumes, see the chapter “Advanced Revelation Volumes” in *Filing Systems*.

When indicating the location of the file, use the path name for the file as understood by your network. For example, if your network has available a logical drive such as “T:” you can simply use that drive location when you attach the file or create a volume. If you are accessing data from multiple servers, you might use a path name something like this:

```
ATTACH ADMIN\SYS:PAYROLL
```

- See your network documentation for information on creating and using path and drive names.

Creating and Deleting Files

You can create a new file on a network any time, just as you can on a stand-alone system. However, before other users on the network can access the new file, they must attach it.

Therefore, when you create a new file that others will share, you should alert them. If they need to use the new file, they can then attach it. Use the facilities of your network to tell others about the new file, including its location.

The same is applicable when you delete a file. As soon as you have deleted a file, alert other users that they should detach the file. If they continue to attempt access to a file that has been deleted, they will experience errors.

File Types

You can store any type of file on a network drive. However, if you intend to share data with other users, you *must* use a file type that can support file and record locking. In Advanced Revelation, the only file type that does this is Linear Hash. Therefore, always choose the Linear Hash file type when defining and creating files for shared access.

Warning! Do not attempt to share data with other users if you are not using a file type that supports locking. If you do, you will experience logical or physical file corruption.

If you are using an Environmental Bond, the file you are using may allow you to share data with other users. For example, the dBASE bond supports locking, so that you can safely share the file with other users on a network. For information on whether your Environmental Bond supports locking, see the documentation that accompanies your bond.

- ☛ To safeguard data in files that do not support locking, you may wish to use your network operating system to set the files' status to "non-shareable". See your network documentation for details. Linear Hash files must *always* be set to shareable using your network operating system, even if you do not intend to share data with other users.

The Advanced Revelation Environment

Because multiple users will be accessing the same copy of Advanced Revelation, it is critical that all users understand how to access a network copy of Advanced Revelation properly. In addition, it is more important to use the built-in facilities for creating unique users and environments.

Logging on to Advanced Revelation

If you intend to access Advanced Revelation files on a network, you should be using a network version of Advanced Revelation. A network version of Advanced Revelation:

- Has been bumped to support four or more users.

- Resides on a network drive.
- Is the same copy of Advanced Revelation that others on the network are using.

You should not attempt to access Advanced Revelation files on a network drive unless your copy of Advanced Revelation has been bumped. If you use a single-user copy of Advanced Revelation to access files, you cannot lock files or records. As a result, your access to files can conflict with that of other users, and the result might be data corruption.

You should also be sure that you are using a copy of Advanced Revelation (the AREV.EXE file) that is on the network. In general, you should log on to Advanced Revelation using this sequence:

1. Log on to your network.
2. Switch to the drive or subdirectory where you store your network copy of Advanced Revelation.
3. Log on to Advanced Revelation.
4. Attach volumes and files that your application requires.

Do *not* use a path name or search drive when you log onto Advanced Revelation. For example, you cannot be on drive T: and invoke Advanced Revelation using the command F:\AREV. .

Make sure that you are using the same copy of Advanced Revelation as everyone else. In other words, do not keep multiple copies of Advanced Revelation on your network. If you and other users are using separate copies of Advanced Revelation, you may not be able to lock files and records correctly, and may experience data corruption.

Finally, you may find that some networks use quite a bit of memory in your PC when you are logged on to them. Be sure that you have at least 512K of RAM available when you attempt to log onto Advanced Revelation. If there is not sufficient memory, you may need to reconfigure your network software (such as your network device drivers) so that you can free up sufficient RAM for Advanced Revelation.

If you have established users in Advanced Revelation (see "Users and Environments" below), you can log on using the user name as part of your AREV command. For example, to log on to Advanced Revelation as the user JOYCE, type this at your operating system prompt:

```
AREV JOYCE
```

When you enter Advanced Revelation in this way, the environment, video, and logon options for the user you specify will be invoked.

Users and Environments

By creating Advanced Revelation users, you can customize Advanced Revelation to the hardware and software requirements of each station on your network.

- ✎ For information on establishing users and environments, see the chapter “Users and Accounts: Planning Access and Security” in the section *Developing an Application*.

There are three types of settings you may wish to customize: environment settings, video settings, and logon scripts. In addition, there are a set of global environment settings that apply to all users on a network.

- ✎ After you change settings for a user, that user should log off and then back on. Many of the settings you can change are only read by Advanced Revelation when you first log on.

Environment Settings

The environment for a user determines such factors as what appears as the main menu, and what help level is active. You can change any setting in the environment to suit an individual user. Particular settings that you may wish to customize for individual users include:

- *Help level.* This determines how much information is displayed in the status line, and whether help messages appear automatically.
- *Menu.* If a user should see an application-specific menu at logon time, change this setting.
- *Default data volume.* If you wish to have Advanced Revelation automatically create and attach files on a particular volume, change this setting.
- *Rollout file.* If you use the SUSPEND command, Advanced Revelation stores information in an operating system file until you use the EXIT command to return. See “SUSPEND” under “Executing Network Commands” later in this chapter.
- *Index settings.* Settings in the Index environment window control how a station will use Advanced Revelation’s background indexing feature. For more information on these settings, see the topic “Managing Indexing on a Network” later in this chapter.
- *Transaction volume.* Using the Concurrency window in the environment, you can establish a separate volume for each user’s transaction files. For more information see “Transaction Processing on a Network” later in this chapter, and the chapter “Managing Files and Records” in *Filing Systems*.

- Video Settings** Some users may have color monitors, while others may be using monochrome monitors. By establishing video settings, you can tailor the look of Advanced Revelation for each user. In addition to setting colors, you can indicate a preference for border types and a background character.
- Logon Scripts** You can use a logon script to attach files, run initialization processes, display menus, or almost any other task. Each user you create in Advanced Revelation can have a different logon script. This is important not only for processing, but for security as well.
- Global Environment Settings** A small set of environmental settings must be the same for all Advanced Revelation users on a network. These global settings cannot be customized for an individual user on the network.
- Global environment settings pertain to transaction processing, to implicit locking in SQL, and to coordinated locking. For information on changing these global environment settings, see “Transaction Processing” in the chapter “Managing Advanced Revelation Files and Records” in *Filing Systems*, and the chapter “Locking” later in this section.
- If you change a global environment setting, *all* users must log off and log back on in order to read these settings.

Executing Network Commands

You can execute network commands from within Advanced Revelation in the same manner that you execute operating system commands such as DIR, COPY, and FORMAT. To do so, use the PC or SUSPEND command at TCL. To return from the operating system to Advanced Revelation, use the command EXIT.

Using the Network Command Processor

As with executing operating system commands, your computer must be able to find and load the command processor for your operating system. For example, in DOS this is established using the command similar to this:

```
SET COMSPEC=C:\COMMAND.COM
```

It is particularly important when on a network that you use the version of the command processor (COMMAND.COM) that is correct for your computer.

- ☛ For information on using the command processor from within applications, see your network documentation.

SUSPEND

If you use the SUSPEND command, Advanced Revelation is rolled out to an operating system file. You can establish the path and file name for this file using the TCL ROLLOUT command or the environmental setting for the rollout file.

Each user on the network should have a unique setting for the rollout file. This will prevent interference with one another if more than one attempts to suspend Advanced Revelation at the same time. If a local drive is available, a user may wish to roll out to that drive.

Printing on a Network

You can print easily to a network printer from within Advanced Revelation. Most networks allow you to redirect your printer output to a printer that is attached to the network. Printing to the network from within Advanced Revelation is therefore a simple matter:

1. Redirect printer output to the network printer.
2. Generate your printer output as you would for a local printer.
3. Indicate to the network that you are done printing. In most cases, it is only when this step is done that your output actually begins to appear on the network printer.

In order to accomplish steps 1 and 3, you must be able to execute network-level commands. See “Using the Network Command Processor” above for details.

An example sequence (this one is for a Novell NetWare network) illustrates how you might print to a network printer:

```
PC EXIT CAPTURE /P0
LIST CUSTOMERS BY CITY NAME ADDRESS CITY (P)
PC EXIT ENDCAP
```

In the example, the first command redirects output to network printer zero (P0). The second command generates printer output, and the third command indicates to the network that you are finished using the printer.

Managing Indexing on a Network

Advanced Revelation's indexing system is particularly efficient when you are running on a network. A big advantage is that the task of maintaining the indexes can be shared among several workstations. You can even devote one or more workstations exclusively to this task for maximum efficiency.

- ☛ For more information on creating, maintaining, and using indexes, see the chapter “Creating and Maintaining Indexes” in *Developing an Application* and “Using Indexes” in *Accessing Data*.

Centralized Index Information

When multiple Advanced Revelation users will be sharing the same data on a network, you must be especially careful to manage the indexing system properly. In order to ensure the integrity of your indexes, *all users must be using the same indexing volume (INDEXING)*.

Distributed Index Updates

When you update a file, you do not immediately update the associated index. Instead, you generate an “index transaction” and log it in a central index file. When your workstation is idle, it takes the opportunity to process these transactions and update the indexes.

- ✎ Using environment settings, you can establish how long your station is idle before it begins processing index transactions. See the chapter “Creating and Maintaining Indexes” in *Accessing Data* for more information.

On a network, all stations log index transactions to this central file. As each workstation becomes idle, it processes the next update in line — even if that station did not generate it. In other words, on a network, all stations share index update duties.

If your station generates many index transactions, but is too busy to stop and update the indexes, you do not need to worry. Another station will process your index transactions and update the index for you.

The only restriction is access. In order to process an index transaction, a workstation must have attached the files involved in the index update. For instance, if you generate an index transaction for the CUSTOMER file, another station must have that file (and its index) attached in order to make the update. If no other station can access the file, your station has to make the update when it becomes idle.

Dedicated Index Workstations

You can take the idea of sharing index updates further and dedicate one or more workstations to the task. When you do this, these stations do no other processing *except* index updates. As a result, indexes are updated as fast as possible.

To establish a workstation as a dedicated index station, follow these steps:

1. Make an assessment about how many index transactions will be generated by all users. For instance, find out how often, on the average, index changes will be logged by users.
2. Select a station to be the dedicated index station. At that station, attach as many files involved in index transactions as possible. This station will be able to make index updates *only* in the files it has attached.

3. At the index station, set the *Time between index checks* prompt in the Index environment window. The value you enter here is a function of how much index traffic you anticipate. For example, if there will be moderate traffic, perhaps your assessment will call for this value to be five seconds. This means that the index station will pause five seconds between each index update operation.

☞ If you set this value too low, the index station will generate unnecessary network traffic.

When you wish the station to resume normal functions, reset the value at the *Time between index checks* prompts to its usual value for your system.

Other Network Management Issues

In addition to the general topics described earlier, there are network issues that pertain to specific areas of Advanced Revelation. These include SQL, Transaction Processing and R/BASIC programming.

Implicit Locking

Advanced Revelation SQL uses “implicit” or automatic locking to control access to files by multiple users. Because much of the decision-making for multi-user coordination is done by SQL itself, there are several environment settings that apply to implicit locking. These are described fully in the chapter “Locking”.

Transaction Processing on a Network

Transaction processing on a network is largely the same as it is on a stand-alone system. However, because multiple users will be using transaction files, you must consider several factors.

- ☞ For more information about transaction processing, see the chapter “Managing Files and Records” in *Filing Systems*.

Transaction Volume First, each user can determine where they wish to store temporary transaction files. These files are used to hold updates until the user commits the transaction.

Temporary transaction files can be located anywhere (local or network), and can be in a different location for every user. Locating transaction files on a local drive can reduce network traffic. To establish a location for transaction files, use the Concurrency environment window.

Central Commit Log

While a transaction is being committed, it is logged in a central commitlog. *The volume on which this log is maintained must be the same for all users on a network.* Because of this, you set the location of the commitlog volume using the Global environment window.

Commit Protection

On a network, multiple users can be committing transactions at the same time. This may mean that several users are committing to the same file at the same time.

By default only one user can be committing against a file at a time. In order to accomplish this, separate queues are maintained for each file involved in a commit process. When a transaction attempts to commit (or restart a commit), entries are placed in the queue for each file updated during the commit. The result is that files are updated chronologically, according to the time when a transaction was committed.

By setting the *Commit Protection* prompt in the Global environment to ON, you specify that you want this one-at-a-time protection. Setting this prompt to OFF boosts commit performance. However, you may sacrifice data integrity, because multiple users can commit at the same time to the same files. There is no guarantee of chronological updates to files.

The COMMITLOG Utility

You can view and maintain the commitlog and associated file queues using the TCL command COMMITLOG. This may be necessary, for instance, if a workstation experienced a failure while in the middle of a commit process. If commit protection is on, other users may be prevented from finishing their commits. The COMMITLOG utility allows you to clear the first user from the commitlog, so other users can proceed.

- ☛ A user in the middle of a commit operation may already have updated data files. If you clear a user from the commitlog, the updates already made by that station are not backed out. As a result, there may be logical data corruption in the files. You will need to manually finish or back out changes made by that user.

R/BASIC Programming for Network Applications

The applications that you write in Advanced Revelation might require programs written in R/BASIC. If so, you must take extra steps in order to ensure network compatibility for your programs.

- ☛ You can write all your programs with the idea that they will be run on a network version of Advanced Revelation. If they are run on a stand-alone PC, the network logic will be ignored.

Locking If your programs will be updating file or records, they should set locks before reading records. You must use the R/BASIC LOCK and UNLOCK statements in your programs — there is no other mechanism in R/BASIC for coordinating file access with other users.

See the chapter “Programming with R/BASIC” and the commands LOCK and UNLOCK in the chapter “R/BASIC Command Reference”, both in *R/BASIC*, for more information.

The system variable @STATION contains the station identifier for your workstation. You can use this number to tag files, records, or processes uniquely on a network. For example, if multiple users might be writing log entries to a file, you can use @STATION as part of the log record to identify which station it came from.

Chapter L 3

Locking

Locking is the process of coordinating data access with other users on a network. This chapter describes what locking is, what types of locking Advanced Revelation makes available to you, and how to establish locking in your applications. In addition, the chapter contains information about how locking is used in Advanced Revelation SQL.

Introduction to Locking	L 3 . 1
Types of Locking	L 3 . 2
Establishing Locking	L 3 . 5
Setting Locking Options	L 3 . 7
Implicit Locking	L 3 . 8
Additional Lock Issues	L 3 . 13



Introduction to Locking

One of the major requirements of a network-ready system is the ability to coordinate the requests of multiple users. To do this, Advanced Revelation allows you to “lock” files and records.

Purpose of Locking

Whenever you are accessing data on a network, there is always the possibility that another user may also wish to access the same data. In order to coordinate multi-user access like this, Advanced Revelation offers a variety of file and record locking methods.

When you lock a file or a record, you are indicating that no one else should attempt to update it. In effect, it is like posting an “in use” sign on the file or record.

While you have the file or record locked, you can update it. When you are done, you unlock the file or record. This clears the lock you have set, and other users may then perform updates.

Locks vs. Data Access

It is important to understand that locking a file or record is separate from accessing it. When you lock a record or file, you are not really physically preventing another user from accessing it. Instead, you are preventing that user from *locking* it. *Other users can still access data while you have it locked.*

This means that in order to coordinate data access, all users must use “locking protocol”. If another user wishes to update a file or record, that user must always first lock the file or record before doing so.

Locking done this way is sometimes referred to as “logical” locking, since there is no physical lock mechanism — only a “semaphore” that is interpreted as a lock. As an analogy, compare how traffic lights work. A red light means “stop”, but it does not physically impede you from going through an intersection. However, in order to prevent accidents, drivers adhere to the “traffic light protocol”, stopping on red and proceeding on green.

It is an advantage that you can still access a locked file or record. For example, suppose that one user has locked a parts file preparatory to applying global price changes. If the file is large, it may remain locked for a long time while the price changes are being made.

Other users, however, may need to look up the names of parts (knowing that the prices are not accurate). Because they do not intend to update the parts record, they might read the record *without* locking it. They are not attempting to lock the record, so their access does not conflict with the file lock established by the user who is updating the prices. If they *did* attempt to lock parts records, they would be refused.

Locking your own Locks

A feature of Advanced Revelation locking is that it can detect if you have already set a lock on a file or record. For example, if you are using the Advanced Revelation editor, you might call the TCL window, and then (perhaps inadvertently) attempt to edit the same record again.

If you already have a file or record locked, a subsequent attempt to lock it again will fail, just as if another user set the lock. This preserves data integrity, since you might otherwise be making different changes from two or more places in the system.

If you are an R/BASIC programmer, you can determine that you have already locked a file or record. When a lock attempt fails, you can test the status of the lock to determine if you or another user has set the existing lock, and proceed accordingly.

Locking in a Single-User Environment

Whether you are running a network or a single-user version of Advanced Revelation, you can always use file and record locking. In a network environment, your locks will be passed through to the network. In a single-user environment, your locks will be ignored, and there will be no effect on your applications.

Types of Locking

When you use locking in Advanced Revelation to coordinate access to data, there are many options. These involve:

- At what level to lock (a whole file or just a record)
- What degree of locking you need (exclusive or shared)

As you use different facilities within Advanced Revelation, you will have the opportunity to choose from among these options. (Some processes do not give you a choice — they are pre-defined to use a particular type of locking.)

- ☞ For more information on when and where you can choose locking options, see “Establishing Locking” later in this chapter.

Levels of Locks

Your first option is in determining at what level to set a lock. You can lock an entire file at once, giving you full control of the file. Alternately, you can lock a single record at a time, and not tie up files for other users.

File Locking When you have a file locked, other users cannot also lock that file. Use file locking when you need to:

- Make extensive changes to a file.
- Conserve the number of locks used.

Since some networks limit the number of locks you can set, it may be impractical for you to lock each record you update individually. If you will need to lock a great many records concurrently (before unlocking them), it is more effective for you to use a file lock.

☞ Other users *may* be able to lock individual records within the file (see “Coordinated Locks” below for details).

Record Locking When you have a record locked, other users cannot lock the record you have locked, but they can lock other records in the file.

Use record locking when you need to:

- Make specific changes to a file.
- Allow multiple user access to a file.

Degrees of Locks

Whether you are locking a file or a record, you can set different *degrees* of locks. By specifying a degree of locking, you can share access to the data, or restrict all other users.

☞ Your network may not support all of the methods described below. Refer to your network documentation for information.

The locking methods operate according to an established priority. You should refer to each particular locking method described and to *Table L3-1* below for information about these relationships.

Exclusive Locks An exclusive lock is the most typical lock. When you set an exclusive lock, no other user can set a lock *of the same level* (file or record) on the file or record you have locked.

If you have applied an exclusive file lock, another workstation may still be able to place an exclusive record lock on a record in the file. See “Coordinated Locks” below for details.

Shared Locks When you have a shared lock set, other users can also set a shared lock. However, you cannot apply an exclusive lock while a share lock is set. Similarly, you cannot set a shared lock while an exclusive lock is set.

You should not use shared locks if you intend to update a file or record. Instead, use a shared lock when you need to read to a file or record (such as for a report), and it is important that no other user set an exclusive lock at the same time.

- ☞ Some networks do not support shared locks. If this is the case, Advanced Revelation will always set exclusive locks.

Coordinated Locks

When you set an exclusive file lock, you are preventing other users from also locking that file. Similarly, when you lock a record exclusively, other users cannot also lock that record.

However, even if you have an exclusive lock set, users may be able lock at a different level. For example, if you have an exclusive file lock set, other users might still set exclusive record locks in that file. In other words, a file and a record lock are may not be “coordinated”.

A coordinated lock sets a file lock, then a record lock. Coordinated locks are used when you need to mix levels of locking.

- ☞ Coordinated locking can be controlled using an environmental setting. See “Using Coordinated Locking” under “Lock Options” below.

Shared Coordinated Locks

A shared coordinated lock first attempts to set a shared lock on the file. If this lock is successful, a shared record lock is then set as well.

- ☞ The shared coordinated lock can only be applied to records.

This type of lock may be used to prevent updates to a record while it is being read.

Exclusive Coordinated Locks

An exclusive coordinated lock first attempts to set a shared file lock. If this lock is successful, an exclusive record lock is then set as well.

- ☞ The exclusive coordinated lock can only be applied to records.

This type of lock may be used to prevent updates to a record when coordinated locking is enabled (the *Coordinated Locking* setting at the Global Environment window).

Lock Table

The table below illustrates all possible combinations of locks, and tells you what locks will be successful when faced with a locked file or record. The attempted lock is listed vertically, with the lock already set listed horizontally.

Table L3-1

Lock Attempted	Lock Already Set					
	XFL	SFL	XRL	SRL	XCRL	SCRL
XFL	N	N	Y	Y	N	N
SFL	N	Y	Y	Y	Y	Y
XRL	Y	Y	N	N	N	N
SRL	Y	Y	N	Y	N	Y
XCRL	N	Y	N	N	N	N
SCRL	N	Y	N	Y	N	Y

Establishing Locking

Whenever you update files or records in a network version of Advanced Revelation, you should be using locks. In some cases, this is taken care of automatically. In other cases, you must indicate your preference for the type and degree of locking you wish to use. If you are an R/BASIC programmer, you must set locks explicitly.

Locking in System Routines

All system utilities that update records already have locking built into them. The type of locking they do is suited to the purpose of the utility. For example, utilities that update records are designed to execute exclusive record locks as they process. In some cases, you may have control over the type or degree of locking that the utility achieves.

- ☛ The documentation for individual commands in Advanced Revelation will indicate the type and degree of locking used by that utility.

The Advanced Revelation system automatically locks a record when you move to a new TCL level. This protects you from accidentally making conflicting changes to your open records. You can still access the locked record(s) from another level or environment, but you cannot save a record you have already locked (see "Locking your own Locks" earlier in this chapter).

Automatic Locking

Many processes that you use in Advanced Revelation will lock file or records automatically. For example, the Advanced Revelation editor will attempt to set an exclusive record lock on the record you are accessing.

If the editor encounters a locked record, you will see a message similar to this one:

```
recordname is in use elsewhere
and cannot be saved

..Press any key to continue..
```

You can view the record at that point, but may not make any changes.

User-Optional Locks

Other processes in Advanced Revelation allow you to option of setting locks yourself. For example, the record copy utility (accessed with the Recordcopy window or the TCL RECORDCOPY command) includes an option to lock records as they are being copied. This is the (L) option.

If you choose this option, the record copy process will attempt to set exclusive record locks on the source and target records as they are being copied. If the record cannot be locked, you will see a message like this one:

```
Waiting until the record recordname
can be locked
```

Advanced Revelation's window processor offers you even finer control over locking. Using the Customize menu option in Paint, you can set a lock preference at the *Locking* prompt.

The lock type you choose at this prompt in Paint will determine how the window will lock records. You can choose shared, exclusive, and coordinated record locks, or no locking at all.

- ☛ For more information on setting locks using Paint, see "Adding Advanced Features to Windows Using Paint" in *Developing Applications*.

Explicit Locking

If you are an R/BASIC programmer, you can take complete control over locking. Using the R/BASIC LOCK and UNLOCK commands, you can specify your preference of file or record, and shared, exclusive or coordinated locks.

In addition, you can take control over what happens when a locked file or record is encountered. For example, you can post messages and wait for user response, skip the update, wait and try again, or devise another strategy for accommodating locks.

- ☛ For information on locking in R/BASIC, see the LOCK and UNLOCK commands in the "R/BASIC Command Reference" in *R/BASIC*.

Implicit Locking Finally, if you are using Advanced Revelation's SQL, you can take advantage of "implicit" locking. When you update files or records using SQL commands, you can set a "consistency level" that indicates how much locking SQL is to do during the update.

- ☛ Implicit locking is discussed in detail under "Implicit Locking" later in this chapter.

Setting Locking Options

In addition to specifying the level and degree of locking, there are some optional parameters you can use to manage locking.

Limiting Available Locks

Networks vary in the number of locks that may be set. For example, a network may set a limit of 400 concurrent locks for all users on the network, or may set a limit based on how much memory is available in the file server.

If you want to conserve or limit the locks available to a workstation, use the *Lock Limit* setting at the Concurrency Control Environment window.

- ☛ Refer to your network documentation for the maximum number of locks available.

Setting *Lock Limit* to 0 (zero) disables lock limiting. The workstation may then use the maximum number of locks available to the network. When the limit for a station is exceeded, the next lock fails. If you programming in R/BASIC, you can detect this error using the system variable @FILE.ERROR.

Unlocking Transactions at Commit

For files with transaction control applied (including those being used with SQL commands), you can choose to release all locks when a workstation commits transactions. Because of the reduced checking and coordination involved, performance improves considerably.

However, *all* locks that you have set are released, whether explicit or implicit, and whether set by the system or by you. The Advanced Revelation system locks records in all environments whenever you temporarily exit the record (to TCL, for example). To avoid the possibility of releasing these locks, the you should not have open records that may be affected. If you wish to guarantee maximum control over unlocking, you should *not* use this setting.

Use the *Unlock All at Commit* setting at the Concurrency Control Environment window to enable the unlock process.

Using Coordinated Locking

You can apply coordinated locking globally, for all workstations, by setting the *Coordinated Locking* prompt at the Global Environment window. When this prompt is set to “Y”, all record locks will automatically attempt coordinated locks. If the prompt is set to “N”, file and record locks are attempted individually, with no checks made for the other level of locking.

☛ See “Coordinated Locks” under “Types of Locking” above.

Implicit Locking

In Advanced Revelation SQL, locking and concurrency controls are automatically (implicitly) applied. Although explicit locking is available throughout Advanced Revelation, implicit locking and concurrency controls are unique to SQL.

☛ Since implicit locking is used exclusively with SQL, the word “table” is used here in place of the word “file”. A table lock is another name for a file lock.

Concurrency Controls

Concurrency controls are a way to establish settings for locking and data integrity using a single parameter. The controls provided in Advanced Revelation SQL enable you to determine:

- Your choice of table or record locking
- Level of implicit locking (consistency levels)
- Deadlock detection and handling
- Lock waiting

Table and Record Locks

The prompt *Implicit Locking* in the Global Environment window determines whether SQL will lock tables or records. If the prompt reads “Record”, records are locked individually before each SQL update. If you enter “Table”, SQL will lock entire tables before updating them.

☛ If you choose record locking, you may run out of locks for your workstation or on your network, especially if you are using transaction processing. See “Limiting Available Locks” above.

Setting Consistency Levels

“Consistency level” refers to your ability to define the level of locking used and the duration of the lock (when locks are released). Five settings allow you to specify a level from no locking at all to a setting that guarantees exclusive access to data for the duration of a transaction.

Control through consistency levels is dependent on four factors. These are:

- Table and record locks. See “Table and Record Locking” above for details.
- A distinction between *update* operations (UPDATE, INSERT, DELETE) and a *read* operation (SELECT). An update operation makes changes to a table, while a read operation simply reads records.
- A distinction in duration between a *query* and a *transaction*. A query is the length of time required to execute any single SQL statement. A transaction remains active until it is committed or rolled back — this may last for the duration of many queries.
- A distinction between shared and exclusive locks. See “Degrees of Locks” earlier in this chapter for information.

You can set consistency levels at the Concurrency Control Environment window. The default consistency level is assigned at logon time based on this value. You can use the SQL command SET TRANSACTION to change the default value for the remainder of the logon session. The following are the settings available for consistency level:

- Level 0* A setting of zero completely disables any implicit locking done by SQL. In order to lock records and tables, you must lock them explicitly using R/BASIC commands. You should not choose this consistency level unless you are working with a single-user system.
- Level 1* A consistency level setting of 1 causes SQL to set exclusive locks during *updates* of tables or records. The lock is held for the duration of the query. No locking takes place during reads.

Operation	Lock Degree	Duration
Update	Exclusive	Query
Read	none	none

- Level 2* A setting of 2 sets exclusive locking during *updates* for the duration of each transaction. When the transaction is committed or rolled back, the locks are released. No locking is done for read operations.

Operation	Lock Degree	Duration
Update	Exclusive	Transaction
Read	none	none

Level 3 A setting of 3 sets exclusive locks during *updates* for the duration of each *transaction*. In addition, it sets shared locks during *reads* for the duration of each *query*.

Operation	Lock Degree	Duration
Update	Exclusive	Transaction
Read	Shared	Query

Depending on whether you are using file or record locking, and if you are using coordinated locking, a consistency level of 3 might provide protection against re-reading records that have changed during your transaction.

Level 4 A setting of 4 sets exclusive table locks during *updates* for the duration of each transaction, and shared table locks during *queries* for the duration of the transaction.

Operation	Lock Degree	Duration
Update	Exclusive	Transaction
Read	Shared	Transaction

This level, the most stringent one, guarantees that other users cannot make changes to any table used or locked by the current transaction. No records can change, and no new records can be added to a table by another user when the consistency level is set to 4.

Because a level of 4 can potentially use a large number of locks, it is only available if you are doing table locks. The *Implicit Locking* prompt at the Global Environment window must be set for table locking to use this level.

You can guarantee that transactions are processed chronologically when *Commit Protection* is set at the Global Environment window. This is the only level of consistency that can provide chronological commit operations.

Detecting Deadlocks

A deadlock situation occurs when two or more workstations each attempt to lock data that the other already has locked. For example, if the first user has record A locked and attempts to lock record B, while the second user has record B locked and attempts to lock A, a deadlock situation is created.

The implicit locking scheme in SQL can determine that you are involved in a deadlock situation. There are options that you can specify as well that determine how the deadlock will be resolved.

The system does not provide any automatic deadlock detection for explicit locking. However, using R/BASIC you can identify and manage deadlock situations as you wish. For more information, see "Programming with R/BASIC" in *R/BASIC*.

- ☞ The deadlock detection process will ignore coordinated locks.

Deadlock Posting

If the implicit locking system suspects that it has encountered a deadlock, it "posts" its own lock information in a central file. This file accumulates information about all workstations involved in the deadlock until the system can determine how to resolve the conflict.

You can specify the amount of time a workstation waits for a lock before posting itself to the central deadlock file. Once this information is posted, the deadlock checking process (see below) begins testing for a deadlock situation.

Use the *Deadlock Posting* prompt at the Concurrency Control Environment window to specify an optimum length of time before the posting occurs. The default value for *Deadlock Posting* is 15 seconds. If you set *Deadlock Posting* to 0, the deadlock checking process is disabled.

Deadlock Checking

After a workstation posts the lock to the deadlock file, the deadlock checking process is enabled. The system checks the deadlock file at specified intervals in order to determine whether a deadlock is indeed in progress.

The *Deadlock Checking* prompt at the Concurrency Control Environment window specifies the time interval used by the deadlock checking process to test whether a deadlock is in effect. The default value for *Deadlock Checking* is 10 seconds.

Deadlock Resolution

When a deadlock is indeed in progress, one of the stations involved in the deadlock must give up its locks and roll back its transaction. The *Deadlock Resolution* prompt at the Global Environment window determines how a workstation determines if it must give up its locks.

- ☞ Each workstation may select only itself as a victim.

The settings you can specify for *Deadlock Resolution* are:

- Least Active
- Youngest

Least Active. The Least Active setting uses an activity index to determine the selection. The index is based on the amount and type of I/O activity, with greater value accorded to write operations. The stations with the lowest activity index becomes the victim, rolling back its transaction and releasing its locks.

✎ The value for the activity index and the current transaction timestamp can be displayed with the TRANSACTION STATUS command.

Youngest. The Youngest setting uses the time stamp for the transactions to identify the most recent transaction and determine the selection. If you choose this method of victim selection, the workstation that began its transaction most recently will roll back its transaction and release its locks.

The Lock Wait Process

Often a workstation will attempt to lock data that is already locked. If a station is attempting to set an exclusive lock, and is continually defeated by the shared locks of other stations, this is called a "livelock". The only solutions in this situation are to wait until the lock is released or to quit the process and roll back the transaction.

Automatically Ending Lock Waiting

You can specify a length of time to automatically end the lock wait process. To do so, use the *Lock Timeout* prompt at the Concurrency Control Environment window.

At the end of this time, the SQL command will be aborted. You then have the option to commit or roll back, ending the current transaction. You can also choose to ignore, which aborts the current command but does not affect the transaction.

If you are using embedded SQL, the command will return with the SQLSTATE value set to indicate a failed lock due to an interruption.

Manually Ending Lock Waiting

Rather than continuing to wait for a lock, you can choose to manually interrupt the lock wait process (unless this is disabled, see below). By pressing [Esc] while waiting for a lock, you display the lock waiting popup. From there you have the same choices as described above.

Disabling the Manual Interrupt

If you do not want a workstation to interrupt the lock wait process, you can disable the [Esc] interrupt. To do so, set the *Disable Lock Interrupt* prompt at the Concurrency Control Environment window to “Yes”. The default value for *Disable Lock Interrupt* is “No”.

Additional Lock Issues

Listed below are additional topics concerning the management of locking in Advanced Revelation.

Coordinating Locking with Multiple File Servers

Advanced Revelation does not restrict lock management to a single file server. However, if you intend to use multiple file servers, be certain that:

- Your network supports lock coordination with multiple servers.
 - You attach all accessible file servers before you first log on to Advanced Revelation.
 - You make sure that all users on all servers are using the same global settings and volumes. It is vitally important that all users be accessing the same INDEXING and Commit Log volumes.
- ☞ Refer to your network documentation for information about attaching, accessing, and locking with multiple file servers.

Locking Data in Bonded Environments

The ability to manage locking in a bonded environment depends solely upon the bond implementation. Some bonds can coordinate file and record locking with other Advanced Revelation users, and some can even coordinate locks with users in the other environment. Refer to your bond documentation for this information.

- ☞ See also “Introduction to Bonding” in *Environmental Bonding*.



Chapter M 1

Introduction to Environmental Bonding

Environmental Bonding is the ability of Advanced Revelation to access data from other environments. Using an Environmental Bond, you can read and update information in other database managers, spreadsheets, word processors, and even on database servers, mainframes, and minicomputers.

This chapter provides you with an overview of Environmental Bonding. Here you will find a description of Environmental Bonding, including information on the advantages of bonding. In addition, the chapter contains background information that you will need in order to make the best use of Environmental Bonding.

Environmental Bonding	M 1 . 1
Tenets of Environmental Bonding	M 1 . 3
Using an Environmental Bond	M 1 . 4
Characteristics of Environmental Bonding	M 1 . 8



Environmental Bonding

Almost every computer product or system you use maintains its data in a unique, proprietary format. If you use two word processing programs, for example, you know that each stores its documents in a special format. More often than not, the formats are incompatible, and neither word processor can read documents created by the other.

This problem pervades the computer world. Each database product uses a unique format to store records, as does each spreadsheet. If you want to access other computer systems such as minicomputers or mainframes, there is yet another layer of complexity to overcome in reading and writing data.

Sharing Data

Normally, sharing data between different products or systems requires importing and exporting data. This process consists of using a special process to make a copy of the data in a new format, one that your product or system can read.

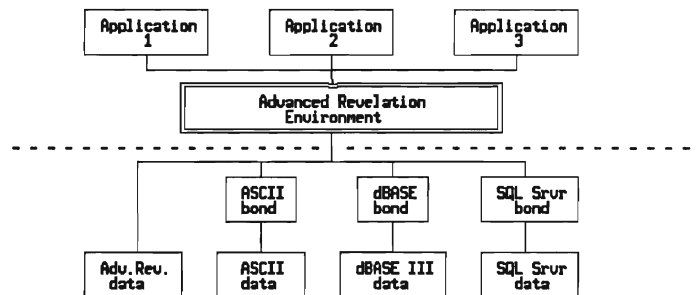
There are several problems with this solution, however. First, importing and exporting data is time consuming. Because you are making a copy of the data, it also occupies extra disk space.

More importantly, if you make a copy of a spreadsheet or database, it becomes difficult to guarantee the integrity of your data. Suppose you export a database to another product. If changes are made in both copies, which copy is current? Unfortunately, the answer is neither.

Advanced Revelation's Environmental Bonding overcomes these problems. An Environmental Bond is a module in Advanced Revelation that gives you direct access to data in its "native" format. Rather than importing the data into Advanced Revelation, you can simply bond to the environment in which the data resides. You can then read and write data directly in that format, sharing it with other users of that environment.

Your applications can use bonded files very much as if they were Advanced Revelation files. An application simply uses Advanced Revelation tools and utilities (windows, SQL, R/BASIC) as always. The environmental bond handles all the work of interacting with the foreign environment, giving your applications a uniform view of data.

The following diagram shows you how Environmental Bonding works:



The direct access offered by Environmental Bonding offers you these advantages:

- There is no time wasted in converting data from one format to another.
- Because you don't copy data from format to format, you save disk space.
- You can use a single set of DBMS tools — including Advanced Revelation windows, R/BASIC, and SQL — to access data in any format and environment.
- Finally, you maintain the integrity of your data. Because there is only one copy of the data, there is no danger that users might update the “wrong” copy of a database.

A New Way to Manage Data

The advantages just listed, however, focus only on technical aspects of data access. In fact, Environmental Bonding introduces an entirely new way of thinking about data management:

- You can take advantage of the capabilities of a specific environment. For example, you can bond to a database server in order to get optimum speed for your queries.
- You open new horizons for your existing applications, which can now access data never before available to them.
- You can even extend the entire concept of “database”. Environmental Bonding allows you to bond to such diverse environments as spreadsheets, word processors, and even CAD drawings. This gives you relational database power in areas that have not traditionally been thought of as databases at all.
- Finally, Environmental Bonding makes the PC the central focus of application development throughout your computer systems. Environmental Bonding allows you to use a single environment — Advanced Revelation — to create and maintain applications for all your computing needs, regardless of where and how the data you require is stored.

Tenets of Environmental Bonding

In the world of information management, there is a vast diversity of file types, data structures, storage methods, access techniques, and capabilities. The challenge for Environmental Bonding is to level out this diversity and overcome the differences between filing systems. Ideally, the result is a completely uniform view of data, regardless of source or location.

To meet this challenge, Environmental Bonding must address a variety of issues.

Intra-Bond Data Integrity

The first goal of Environmental Bonding is to share data transparently with foreign environments. Given any foreign database, an Advanced Revelation application can read each record, write it back without making changes, and result in a database that is indistinguishable from the original. Files updated through an Advanced Revelation bond can be used by applications in the foreign environment as if the changes had been made in the foreign environment.

If this level of transparent update is not possible, the bond should tell the user why a particular update operation would have resulted in data loss and therefore was not performed. Alternatively, the user can request that Advanced Revelation update the file anyway, recognizing that there will be some data loss as a result.

Inter-Bond Data Integrity

A second goal of Environmental Bonding is to guarantee that data can be moved between any two file types without data loss. Given any two foreign environments, an Advanced Revelation application can copy a record from one environment to the other and then back to the original file. The data should be indistinguishable from the original, even by applications in the foreign environment.

If data cannot be copied without data loss (the target file might not accept a field due to its length, for example) the bond must tell the user why a particular update operation would have resulted in data loss and therefore was not performed. As with intra-bond integrity, the user can override Advanced Revelation and request that the file be updated even if data might be lost.

Intelligent Data Access

If a foreign environment supports high-level access methods such as SQL, the bond to that environment is written to take advantage of them. For example, bonds to SQL-based database servers can pass SQL queries to the server for processing.

No matter what high-level access methods are available, a bond always includes the ability to read and write data one record at a time. If the bond cannot use a higher-level access method, it will automatically revert to traditional record access, without intervention by the user.

Transparent File Creation

Using Environmental Bonding, it is possible to write an Advanced Revelation application in such a way that it can create the files it requires in any foreign environment. The files can be used within the foreign environment as if they had been created in that environment.

If a file cannot be created, Environmental Bonding will tell the user which of the requirements could not be supported by the foreign environment (for example, maximum field length or lack of numeric precision).

Using an Environmental Bond

The steps described below provide very general guidelines for using any Environmental Bond. Review the steps carefully so that you will understand what you will need to do in order to access data in foreign files.

- ✦ Always carefully read the documentation that accompanies your bond. Although the steps listed below are general and apply to all bonds, the documentation for a specific bond contains valuable additional information that you will need before using that bond.

Obtaining a Bond

You must obtain a separate bond for each environment to which you want to bond. To list only a few examples, there are separate bonds for dBASE, ASCII, Lotus 1-2-3, and SQL Server files.

Some bonds have been developed by Revelation Technologies. Included in the 2.0 release of Advanced Revelation are bonds for dBASE and ASCII files. In addition, bonds for other environments are available separately.

Other bonds are available through third-party developers. Developers have created bonds to a wide variety of environments, from PCs to minicomputers and mainframes. Information about third-party bonds is available through Revelation Technologies.

If you are a developer, you can also create your own bond. Revelation Technologies will make available to developers a bonding kit containing documentation on Environmental Bonding technology, as well as sample bond programs.

Installing and Configuring a Bond

Once you have obtained the bond, you must install it onto your Advanced Revelation system. There may also be additional steps required in order to configure your system properly to use the bond.

Installing the Bond

Most bonds can be installed using the installation procedure for optional modules. For information on the installation procedure, see the chapter “Updating and Enhancing your Advanced Revelation System” in the section *Installation*.

Bonds available from third-party developers may use different installation procedures. Be sure to read the documentation that accompanies your bond for specific installation instructions.

Configuring the Bond

Some Environmental Bonds require additional steps beyond simply installing the bond. You may need to make changes to existing records in your system, or you may need to know about alternate logon or other options. Be sure to review the bond documentation carefully for information about additional configuration requirements for your bond.

Displaying Installed Bonds

If your bond uses the standard Advanced Revelation optional module installation procedure, you can easily get a listing of what bonds are installed. The Who window displays a list of all the optional modules you have installed, including any bonds. To display the Who window, enter WHO at the TCL prompt.

Establishing Volumes

Because Advanced Revelation can use so many different file types, it must have a way to distinguish them when accessing data. The method used is to designate a “volume” when first accessing a file.

- ☛ Remember that a “file type” identifies the structure or access method of a file. Examples of file types in Advanced Revelation include Advanced Revelation Linear Hash, Advanced Revelation ROS, plus the bonded files types dBASE, ASCII, and others.

A volume is actually two pieces of information in one: the file type and the location of the data. You can create any number of volumes to designate all the combinations of file types and locations that you require for your applications.

- ☛ You can find a detailed description of volumes in the chapter “Advanced Revelation Volumes” in the *Advanced Revelation Filing Systems* section of the documentation. More information about file types appears in the chapter “Files and Records”.

Before using a bond to define a new file or to access an existing file, you must initialize a volume. Follow these steps:

1. Use the options **Tools-Files-Set Volume** to display the Set Volume window.

2. At the *Volume Name* prompt, enter a name for the volume you are establishing. Do not use spaces in the name.
3. At the *File Type* prompt, enter the name of the file type or bond you are using. Press [F2] to see a list of available file types. For example, if you will be accessing data in a dBASE file, enter DBASE_BFS.
4. At the *Data Location* prompt, enter the name of the location where your files are or where you want to create them. In most cases, this will be an MS-DOS or OS/2 drive or path name. In the case of bonds to remote servers, this may specify the database you want to access.
5. At the *Control Files Location* prompt, enter the drive or subdirectory name where you want to store supporting files such as dictionaries. Under most circumstances you can skip this prompt and allow the bond to create the files in a default location.

Control files are described in detail under the topic “Control Files” later in this chapter.

6. Press [F9] to save your volume specifications.

Whenever you use this volume name, Advanced Revelation will know both the location and file type of the files you are accessing.

Defining New Files

Once you have established a volume for bonded files, you can define new files. The bond creates files in the format of the environment it is designed to access. For example, if you define a new file using the dBASE bond, the new file will be a dBASE file.

The file definition process for a bonded file is virtually identical to that for an Advanced Revelation file. You can define new files in a variety of ways. A common way is to use the Definefile window or the DEFINEFILE command at TCL. The Definefile window is described in the chapter “Files and Records” in *Filing Systems*. The DEFINEFILE command is documented in the “TCL Command Reference” chapter of *TCL*.

You can also use Advanced Revelation’s Paint utility to define a new file. Paint lets you create an entry window at the same time that you are defining a new file. Paint is described in detail in the section *Developing an Application*.

If you use SQL, you can use the CREATE TABLE command to define a new file. The CREATE TABLE command is documented in the *SQL* section.

Although there are different methods to define and create a file, the end result is the same. Choose the method that you prefer. For example, if you are unsure of all the options you have when defining a file, use the Definefile window. If you are familiar with SQL, you might choose the SQL CREATE TABLE command instead.

Regardless of what tool you use to define a new file, the following steps describe *generally* how to define a bonded file. Be sure to read the documentation about each tool for specifics about defining a new file.

1. Locate and display the appropriate file definition tool. The Definefile window is available through the options **Tools-Files-Definefile** from the Advanced Revelation Main menu. Paint is available through the options **Design-Window** from the Main menu.

You can enter SQL commands at the SQL window. Remember, you can use the SQL Assistant if you do not remember the syntax for CREATE TABLE.

2. When you are prompted for a volume (for SQL's CREATE TABLE command, this is the ON prompt), fill in the name of a volume associated with your bond. In most cases you can press [F2] to see a list of available volumes.
3. Some bonds may allow you to specify a set of "file attributes". The attributes allow you to indicate bond-specific options for your files.
4. Fill in information about the fields you are defining in the file. This typically includes the names of the fields and their data types. Use the [F1] key to display online help, and [F2] for the available options at any prompt.
5. Save your new definition by pressing [F9].

Attaching Bonded Files

Whether you have defined new files or are using pre-existing ones, you must "attach" the files before accessing them. (If you have just finished creating the file, it is already attached, but you will need to attach it the next time you log on to Advanced Revelation.) As with defining a file, this process is the same for bonded files and Advanced Revelation files.

You can attach all the files on a volume, or individual files. To attach a volume or list of files, use the TCL command ATTACH, or follow these steps:

1. Display the Attach window using the options **Tools-Files-Attach** from the Main menu.
2. At the *Volume* prompt, enter the name of a volume you created earlier for the bond. Press [F2] to see a list of volumes that are available.
3. If you intend to attach only specific files, you can enter the file names at the *File list* prompt. Press [F2] to see a list of files that are available on that volume.
4. Press [F9] to attach the files.

Once you have attached the files, you can access them by name using any Advanced Revelation process. The files remain attached until you log off, or until you explicitly detach them.

Accessing Data in Bonded Files

You can use any Advanced Revelation utility to access data in bonded files. This includes windows, the report generators, SQL, the Editor, and R/BASIC.

One of the great advantages of Environmental Bonding is that from within Advanced Revelation, most bonded files look and act like an Advanced Revelation file. When you are using a window, an R/LIST or SQL report, or a program, it will not be apparent to you that the data is actually stored in a bonded file.

Under some circumstances, you might encounter behavior in a bonded file that makes it different from an Advanced Revelation file. The situations in which this might occur are discussed later in this chapter under “Features and Characteristics of Environmental Bonding”.

Using Additional Bond Options

Besides the basic functions of a bond — reading and writing data to files — an Environmental Bond may offer you additional options. For example, a bond may include extended index functions, downloadable processes, or other features not available through the usual file access methods.

You can access any additional options for a bond at the Bonds menu. To display this menu, use the options **Tools-Bonds** from the Main menu. The Bonds menu has one entry for each bond that supports additional options.

Characteristics of Environmental Bonding

As noted above, one of the features of Environmental Bonding is that a bonded file behaves very much like an Advanced Revelation file. As you work with Environmental Bonding, however, you may encounter differences in features and characteristics for each file type.

The section below describes the areas in which the behavior of a bonded file might diverge from that of an Advanced Revelation file. Because each bond is different, the discussion is necessarily general. For more detailed information about the topics that appear below, be sure to review the documentation that accompanies your bond.

Sources of Differences

There are two reasons why an Environmental Bond might exhibit behavior different from that of an Advanced Revelation file:

- The architecture of the foreign environment dictates that the bond must behave a certain way.
- The developer who implemented the bond has made conscious choices about the characteristics of the bond.

Differences in Architecture

There may be certain Advanced Revelation features that are simply not available in the foreign environment. For example, some file types impose a record-length limit that is considerably less than the 64K available to Advanced Revelation records. Others may not support variable-length fields, while

still others may have a limit on how many fields can be defined in the dictionary.

An environmental bond will always alert you if you have encountered a difference of this sort. Normally, any operation you attempt that conflicts with the capabilities of a bond will produce an error. You may have the option of indicating in advance that you will allow certain functions, such as the truncation of data that is too long for a field or record.

Because these differences are inherent in the environment you are bonded to, Advanced Revelation cannot overcome them. You should therefore be aware of possible differences between the foreign environment and Advanced Revelation before bonding files.

Conversely, of course, a bond may offer features that you do not normally have in Advanced Revelation. For example, some file types allow records even greater than 64K. Others may offer extensions to indexing or other features. Depending on the bond and the feature, these may or may not be available to you within Advanced Revelation.

Differences in Design

Another source of differences between bonds lies not in the file type itself, but in how it was developed. When a developer creates a bond, he or she is free to implement an elaborate bond or a simple one.

An example is a read-only bond — one that allows you to read data from a file, but not update records. In this case, the fact that you cannot write a record is not inherent in the file type, but in how the developer chose to create the bond.

There are perfectly legitimate reasons to create bonds with these characteristics. In some cases, the developer has a special purpose in mind for the bond, and does not require that the bond be capable of all operations.

Accounts

Whenever you create an Advanced Revelation file, it is associated with an account. This is usually the current account when you create the file.

Files from other environments are not *initially* associated with a particular account. However, because accounts are required for all files in Advanced Revelation, Environmental Bonding assigns an account to a bonded file using these rules:

- If the bonded file is created within Advanced Revelation, it is assigned to the current account.
- If the bonded file already existed in the foreign environment when it was first attached, it is assigned to the account GLOBAL.

File and Field Names

When you name a file or field in Advanced Revelation, there are few restrictions on creating the name. You can use any combination of numbers and uppercase alphabetic characters, plus the punctuation marks “.” (period) and “_” (underscore).

- ✎ You can find information about file and field name conventions for Advanced Revelation in the “Managing Files and Records” chapter of *Advanced Revelation Filing Systems*.

Foreign File and Field Names

When you use Advanced Revelation tools to create bonded files and fields, you are free to name them anything you like (within the constraints of Advanced Revelation naming conventions). However, this may conflict with rules that a particular bond has about names.

For example, in some cases the name of a file is limited to eight characters. This is usually the case in systems that use operating system files directly in their applications. There are sometimes similar restrictions on field names.

To avoid a conflict like this, an environmental bond will create or suggest an alternative foreign name for the file or field. This name will fall within the naming rules for the bond you are working with.

You can override this suggested name with your own foreign name. However, your name must still follow the naming rules for that bond.

When you have assigned a foreign name, the file or field will then have *two* names:

- The name that you can use while accessing the file or field within Advanced Revelation.
- The foreign name by which the file or field is known in the foreign environment.

Control Files

When you bond to a file from another environment, Advanced Revelation creates and maintains one or more “control files”. These files are required by Advanced Revelation, but are not used by the foreign environment.

One type of control file is a dictionary. No matter what type of environment you bond to, Advanced Revelation will always create and maintain an Advanced Revelation-style dictionary for each bonded file. Another control file is the volume directory that Advanced Revelation maintains for each volume.

If you establish Advanced Revelation indexing for your bonded file, the index will also be a control file. However, if you establish an index using the capabilities of the

foreign environment, the index is *not* considered a control file — instead, it belongs to that foreign environment.

For example, if you create an index for a dBASE file using the dBASE SET INDEX command, the index you create will be a part of the dBASE environment. dBASE users will be able to use and maintain the index. Conversely, if you use standard Advanced Revelation indexing tools to create an index for an ASCII file, the index is internal to Advanced Revelation and is considered to be a control file.

- ☛ See the bond documentation for your bond to learn more about the indexing capability of the bond.

Specific bonds may also maintain additional control files. For information about the files and records that a bond creates and uses, see the documentation for that bond.

Because the control files for a bond are internal to Advanced Revelation, they are created using the Advanced Revelation file type. This means that when you access data in a bonded file, you will actually be using *two* file types: the foreign file for the data, and Advanced Revelation files for the dictionary and indexes. This is all transparent to you, however, as you use your bond. The bond itself will handle details of coordinating access to all the files it requires.

Volumes and Control File Volumes

Because bonded files are not the default file type in Advanced Revelation, you must initialize a volume in order to access them. To do so, use the Set Volume window to create a new volume and to indicate the location and file type of the files.

- ☛ For information on creating volumes, see the chapter “Advanced Revelation Volumes” in *Advanced Revelation Filing Systems*. Establishing a volume for a bonded file is also described briefly earlier in this chapter under “Using an Environmental Bond”.

Environmental Bonds also typically require that you indicate a separate location for control files. If you use Advanced Revelation files, these control files reside in the same location as the data file itself. Bonds, however, allow you to specify an alternative location for these files.

The intent of specifying an alternative location is to avoid mixing the control files with the data files. Because the control files are Advanced Revelation files, you might not want to store them in the same location as files of a different file type. For example, you might not want to store Advanced Revelation control files in the same subdirectory as your dBASE files.

In fact, you may not even be able to store them in the same location as your data files. If you are accessing a read-only file, for instance, the foreign environment may not allow you to create new files in the same location.

By keeping your data and control files separate, you do not introduce Advanced Revelation files into a foreign environment. Separate control files also make it easier for you to make independent backups of your data files and control files.

You can choose any location for your control files. Normally, each bond will assume a default location, usually a subdirectory beneath the current directory. Wherever you keep your control files, remember to back up that subdirectory or drive if you make a change to a dictionary, index, or other control file.

Volume Directories for Bonds

One of the control files that a bond maintains is the volume directory. This tells Advanced Revelation what files are available on that volume.

A bond builds and maintains the volume directory automatically. Each time you access the volume (by listing the directory or by accessing files on that volume), the volume directory is updated. That way your bond will always know if a user in the foreign environment creates new files or deletes existing ones.

File and Record Structure

Advanced Revelation files and records are quite flexible, allowing you to store large quantities of data in variable-length fields. Bonded file types, on the other hand, may not always offer this level of flexibility.

No matter what differences there may be between Advanced Revelation and the bonded environment, the bond will usually mediate any conflicts or incompatibilities that arise. If there is a potential for loss of information, the bond will verify that you have given “permission” for this to happen.

- ☛ It is up to each individual bond to protect you against data loss. Check your bond documentation for information on how the bond handles potential data loss.

Fields and Records

Many bonds feature a length limitation on records that is well below Advanced Revelation’s own 64K limit. Record length limitations can range anywhere from 255 bytes to 4K or more. Some bonds actually allow records greater than 64K, although Advanced Revelation cannot take advantage of this feature.

It is also common for bonded file types to be fixed-length in some or all of their data types. Some popular file types, for instance, use fixed-length fields for most data types, but allow you to create a special “text” or “notes” field that can contain variable amounts of data.

Most bonded file types do not directly support multivalued fields, as Advanced Revelation does. On the other hand, most will permit you to store data that contains multivalued delimiters in a text-type field.

When constructing or accessing records and fields in a bonded file, it is important that you are aware of the characteristics of that file type. For example, if you can construct a text-type field, you will want to know if you can search that field using string searches (for example, “with the word ‘backorder’ in the comments field”), or if you can store multivalued data in it.

As another example, in some environments a text-type field automatically occupies 4K, whether there is data in it or not. Information such as this may be important to you in designing your applications.

Files Advanced Revelation file types use a “hashed key” method of storage. This method is very efficient for random access of data, and allows Advanced Revelation to maintain variable-length data.

Other file types use different methods for storing and retrieving data. Of course, one of the major goals of Environmental Bonding is to accommodate the differences in the ways various environments store data. The result is that bonded files are treated as if they were Advanced Revelation files, with all differences resolved behind the scenes.

However, because of differences in file structures, bonds do not always perfectly emulate Advanced Revelation files. The topics that follow address differences between bonded files and Advanced Revelation files of which you should be aware.

Implicit Record Keys To retrieve a record from any file in Advanced Revelation, you must always have a “primary key” for the record. This is a field in the record, such as a social security or customer number, that is guaranteed to be unique for all records in the file. An explicit key of this type is actually a part of the record. It cannot change and still identify the same record.

Many file types, however, do not use a primary key. Instead, they use an “implicit key” that describes the record’s position within the file. Rather than retrieving a record using data such as a social security number, you retrieve the record using a designation such as “the hundredth record”.

The difference between implicit and explicit keys presents several issues for Environmental Bonding. First, any file type that uses an explicit key can choose which field to use as

the key, and the data type of the key field. With implicit keys this is not possible. An implicit key can usually only be an integer data type, and most often, you are not allowed to specify which field is to be the key.

A more important difference is that while an explicit key never changes, an implicit key *can* change. This might happen, for instance, if a record shifts position in the file because the file was “packed” by a user in the foreign environment.

Because of this, processes that depend on fixed keys can be unreliable. A good example is an index. Because an index stores a record’s key, it is out of date as soon as the key changes. For this reason, you may want to refrain from using Advanced Revelation indexes with file types that depend on implicit keys.

- ☞ Refer to the documentation for your bond to see if the bond uses implicit keys, and how they are handled within the Advanced Revelation environment.

Dictionary

No matter what type of file you access, there is always an Advanced Revelation dictionary for the file. In the case of bonded files, this dictionary is created automatically by the bond.

To the extent that it is possible, Advanced Revelation builds dictionaries according to information available in the foreign environment. For example, the dBASE bond reads the structure of a dBASE file and constructs a corresponding Advanced Revelation dictionary.

- ☞ For some bonds, this is not possible. An ASCII file, for instance, does not store information about its structure. As a result, Advanced Revelation cannot directly construct a dictionary for the file, because there is no way for it to deduce the file structure automatically. In that case, the user is responsible for creating a dictionary structure that matches the file.

Synchronizing Dictionaries

It is always possible that the structure of a foreign file can change from outside of Advanced Revelation. A dBASE user, for instance, might change the structure of a dBASE file. If so, the Advanced Revelation dictionary for that file is no longer accurate.

To account for this possibility, a bond usually checks the correctness of the Advanced Revelation dictionary for a bonded file against the actual structure of the file. This check is done when the file is first accessed.

Synchronization Error

If the structure of the foreign file has been changed outside of Advanced Revelation, you will get an error when you attempt

to access the file. A message will display indicating that the structure of the file has changed since the last time you accessed the file inside Advanced Revelation.

To correct this problem, you must delete the Advanced Revelation dictionary for the bonded file. This will clear the Advanced Revelation view of the bonded file. When you next attempt to access the file, a new dictionary will be built automatically by the bond.

If you want to preserve certain fields out of the dictionary, you should copy them to a temporary file before deleting the dictionary. For example, if you have created some symbolic fields to use with your bonded file, you should copy these to a dictionary file elsewhere in your system. After the new dictionary is built by the bond, you can copy the saved fields to the new dictionary.

Deleting a Dictionary

Because a bond checks the Advanced Revelation dictionary against the foreign file structure when it accesses the foreign file, you will not be able to delete an Advanced Revelation dictionary as you can for Advanced Revelation files. If you delete the dictionary using the syntax `DELETEDFILE DICT filename`, the dictionary will indeed be deleted. However, as soon as you access the bonded file again, the dictionary will be rebuilt as part of the synchronizing process.

Field Position Gaps

One of the attributes that you specify for any type “F” field is the position that the field holds in the record. Normally, you would assign positions sequentially — the first field is position 1, the second field is position 2, etc.

If you are defining an Advanced Revelation file, you can define fields in any order you like. You can also assign positions arbitrarily. For example, you might have fields 1 through 3, skip position 4, and then continue with fields 5 through 11.

Bonded files, however, rarely allow this flexibility. Especially if you are defining a fixed-length file, fields must be ordered sequentially with no gaps.

Environmental Bonding will enforce this restriction if it applies to your bond. If you are using Paint or the Dictionary window, it is possible to create fields with non-sequential positions. But when you try to create the file, the bond will refuse to do so. You must then edit the dictionary definitions, and make sure that they are all sequential.

- ☛ Check your bond documentation to see if the bond allows you to leave “gaps” in your record definition.

Restructuring Files

Because of the variable-length nature of Advanced Revelation files, there is normally no restriction on making changes to the structure of a file. For example, you can add a field at any time by simply creating a new record in the dictionary.

Many other file types do not allow this, however. Typically, once a file is created, you cannot add new fields or remove existing ones.

To change the structure of these types of files, you must completely rebuild the file. As a rule, Advanced Revelation leaves this to the foreign environment. For example, if you want to rebuild a bonded dBASE file, you must do so in the dBASE environment. When you reattach the dBASE file in Advanced Revelation, you can delete and recreate the Advanced Revelation dictionary for that file.

Environmental Bonding will always check a bond to see if you are allowed to make changes to the file structure. If you are not, the bond will prevent you from making the changes.

Warning! If you use Paint to change or add field definitions with a bond that does not allow you to modify the file structure, Paint will prompt you to delete the old file and create a new one. *This does not rebuild the file.* When you rebuild the file, all the data remains intact. If you use Paint to change the structure of a file, *all the data is deleted, and a new (empty) file is created.*

Deleting Files

Because an environmental bond allows you to share files with users outside of Advanced Revelation, there are some guidelines that apply when you want to delete a file.

One of the attributes that is stored for every file is a “delete flag”. You can see and change this flag in the Definefile window at the *Allow File Delete (Y/N)?* prompt.

When the delete flag is “Y,” you *can* delete the data portion of the file.

The setting of this flag is determined when you create a file, or when you first attach it. Advanced Revelation uses these rules when setting the flag:

- If you create the file within Advanced Revelation, the flag will allow you to delete the file (the flag is set to “Y”).
- If you attach a file that already exists in the foreign environment, the flag prevents you from deleting the file (the flag is “N”).

You can change this flag if you like. For example, you might want to delete a dBASE file that already existed when you first bonded to it. To do so, use the Definefile window to set the flag to “Y”, and then delete the file using the Deletefile window or the DELETEFILE command at TCL.

Data Types

Advanced Revelation provides a basic or standard set of data types for fields. These include INTEGER, CHAR, DATE, TIME, and the other data types most often available in database systems.

Each bonded environment, of course, maintains its own set of data types. Some environments may offer additional data types or variants on Advanced Revelation's standard types. Other environments may offer a more restricted set.

In order to be compatible with as many environments as possible, Environmental Bonding matches or "maps" the standard data types of Advanced Revelation against those available in the foreign environment. There are three ways that Advanced Revelation and foreign data types are matched:

- A foreign data type may map exactly to an Advanced Revelation data type.
- If the foreign environment supports fewer data types than Advanced Revelation, an Advanced Revelation data type may map to more than one foreign type. For example, in the dBASE bond, the Advanced Revelation type FLOAT may map into either a dBASE N (Numeric) or a dBASE C (Character) type, depending on the user's choice.
- The foreign environment may have more data types available than Advanced Revelation. In that case, the user has a choice of foreign data types onto which to map an Advanced Revelation data type.

Type Mapping

One of the characteristic features of a bond is how it maps its data types to Advanced Revelation data types. For many applications, it is important to know the nature of data types in both environments, in order to design and implement applications correctly.

When you create a new field in a bonded file, you are actually mapping an Advanced Revelation data type to a foreign data type. For example, if you select the Advanced Revelation data type INTEGER for a field, it will map to the dBASE data type N (numeric).

You do not actually have to make the correlation between the Advanced Revelation and bonded data types. This is done automatically by the bond. You only need to choose the data type appropriate to Advanced Revelation.

- To see what foreign data type has been assigned to a field, you can look in the Dictionary window. Follow these steps:
1. From the Advanced Revelation Main menu, choose the options **Tools-Files-Dictionary** to display the dictionary window.
 2. Enter the names of the file and field.
 3. Press the [Shift-F5] Softkey to display additional field options.
 4. The entry at the *Foreign data type* prompt will tell you what foreign type has been assigned to the field. This prompt is for information only; you cannot change the entry.

Data Type Definitions

A one-to-one mapping of data types between environments is not a guarantee of compatibility. For example, the existence of a type INTEGER in both Advanced Revelation and in the foreign environment does not mean that an integer value will transfer successfully between environments.

It is important to know the *precise* definition of a data type in each environment. For example, in some environments, an integer is defined using 32 bits. In other environments, an integer uses 64 bits. Transferring an integer from the more precise to the less precise environment can result in some degree of data loss (see “Loss of Length and Precision,” below). As another example, the exact definition of VARCHAR in each environment can be quite different.

The documentation for each bond gives you specific information on how each data type is mapped into the Advanced Revelation standard data types. It also lists any bond-specific data types and how these are implemented both in the foreign environment and in Advanced Revelation.

**Advanced Revelation
Data Types**

In Advanced Revelation, all data types are eventually reduced to one of the three following types of data:

- An integer with 64-bit precision.
- A floating-point number with precision to 18 places.
- A character with strings up to 64K.

The following table lists the Advanced Revelation standard data types:

Table #M1-1
Advanced Revelation Standard Data Types

Type	Data
BOOLEAN	integer
CHAR	char
DATE	integer
DATETIME	float
DECIMAL	integer
DOLLARS	integer
FLOAT	float
INTEGER	integer
TIME	integer
VARCHAR	char
TEXT	char

The definitions of each of the types of data in Advanced Revelation dictate the highest possible precision and size available for a bond. If data in a foreign environment exceeds these definitions (for example, a string longer than 64K), one of two options is available:

- If data precision is important, the read or write operation that encounters the data will fail.
- You can establish permissible data loss, such as a truncated field, in order to accommodate the bonded environment. Details on potential data loss are discussed below.

**Bond-Specific
Data Types**

When you choose a data type for a field in a bonded file, you always have the choice of Advanced Revelation data types. For example, no matter what environment you are bonded to, you can always select the data type DATE or INTEGER. The bond will make sure that the data type you select is mapped appropriately to the foreign environment.

Bonds frequently add their own data types to the list of types available for a field. You can always tell when a data type is bond-specific, because the name of the bond precedes the type name. For example, the data type D (date) from the dBASE environment appears as the data type DBASE_D within Advanced Revelation.

Bond-specific data types give you the opportunity to choose a foreign data type directly. This is useful if you are already familiar with the data types available in that environment. In that case, you do not need to examine how an Advanced Revelation data type maps to the foreign data type. For example, if you are a dBASE user, you do not need to choose between the Advanced Revelation types INTEGER and

FLOAT — instead, you can directly choose the dBASE type N (numeric).

The documentation for a bond always describes any bond-specific data types. This documentation provides you with information on the names, characteristics, and mappings of these types.

Bond-specific data types are automatically added to your Advanced Revelation system when you install a bond. You can therefore see a list of these data types whenever you are asked for data type. For example, if you are defining a dBASE file and are asked for a data type, the [F2] Options popup will list not only the Advanced Revelation standard data types, but the dBASE-specific data types as well.

- ☛ Remember that the Advanced Revelation data types are generic and are implemented in each bond. If portability of your application is an important issue, you will want to use Advanced Revelation data types when defining a field.

Field Definitions

In Advanced Revelation, the dictionary is the central repository for all information about a field. Everything from the field name and size to indexing information to user-specified attributes of a field are stored in the dictionary.

Several of these attributes are specific to bonding. If you use only Advanced Revelation files, you will not normally use these attributes. However, when you create fields in a bonded file, you will need to provide — or at least know about — this additional information.

Master Field

In Advanced Revelation files, you can create any number of fields that describe the same data. As an example, you might create two fields that define field 1 to be a number. The first field might print the data as money, with a decimal point and a dollar sign. A second field might treat the number as a simple decimal number.

When you create fields in a bonded file, you can create these synonym fields also. However, one of the fields that describes the data must be designated as the “master field”. The master field contains not only the usual information about the field (name, display length, justification), but includes all the additional information needed by the bond to access the data (data type mapping, truncation flags, etc.). By allowing only one master field, a bond prevents you from creating two or more fields that have contradictory information about how to access the bonded data.

Creating a Master Field

The first definition that you create for a field will become the master definition. This is true whether you use the Definefile window, the Dictionary window, Paint, or SQL to define your fields. If you define additional fields that describe the same data, they will not be considered master fields — they will be synonyms for the master.

Field Attributes

Each bond can maintain any information it requires about a field. Some foreign environments define relatively simple fields, with few attributes. Others, like Advanced Revelation itself, maintain an extensive list of attributes.

For Advanced Revelation files, the information normally stored in a dictionary definition is adequate to completely describe a field. However, in bonded files, you must almost always include additional information.

To account for the instances where additional information is required, each bond can keep a bond-specific set of attributes in the dictionary. This is a simply list of one or more parameters required in order to completely define the field.

For example, in a fixed-length field, one of the field attributes might be the physical length of the field. The ASCII bond, for example, uses field attributes to store the input and output formats of the data.

You can specify field attributes in the Dictionary window. To access them, follow these steps:

1. From the Advanced Revelation Main menu, choose the options **Tools-Files-Dictionary** to display the dictionary window.
2. Enter the names of the file and field.
3. Press the [Shift-F5] Softkey to display additional field options.
4. At the *Foreign field attributes* prompt, press [F2] to display a field attribute window. You can fill in the field attributes according to the prompts you see. Press [F1] for help, and [F2] for options at any prompt.
5. Save your definition by pressing [F9] (Save).

Loss of Length and Precision

Because each environment has a different definition for its own data types, transferring data from one environment to another can result in loss of information. For example, one environment may define an integer in 64 bits, another in 32 bits. Copying an integer from the first environment into the second results in loss of precision and can therefore result in the loss of some data.

The same applies to character fields. If one environment allows 4K strings, but another is limited to 255 characters, copying data from one to the other can lose data.

Default Protection against Data Loss

Environmental Bonding is designed to fail when you attempt operations that cause any loss of data. For instance, if you attempt to write a field that contains control characters into an environment that does not allow these, the write operation will fail with the error that you have violated a data type restriction.

Degrees of Data Loss

Although Environmental Bonding will not by default allow data loss, you can override this. When you define fields for your application, you have the option of specifying the degree of data loss that you will accept.

- ☛ Specifying a degree of acceptable data loss does not automatically mean that data will be lost. It simply tells the bond what to do if *a particular record* contains data that is too precise or too long for the target environment.

Environmental Bonding allows you three levels of acceptable data loss:

- **None.** You will not accept any degree of data loss. If an operation is attempted that would result in any data loss, the operation is rejected. This is the default for a new field.
- **Some.** Although the preciseness or the length of the field in one environment does not match that in another, the degree of data loss that this represents to you is acceptable. For instance, one environment may use 18 decimal digits for a floating point number, while another uses only nine. You might decide that it is acceptable for digits to the right of the decimal point to be rounded, resulting in a partial loss of information.
- **Total.** You will tolerate the complete loss of information. If you attempt to store extremely large numbers in a field that is not large enough, the resulting truncation may result in total information loss.

When you specify a degree of acceptable data loss, you do so for two conditions: input and output.

Input means data that is being read into Advanced Revelation. If you encounter a potential data loss on input, it implies that Advanced Revelation supports less precision or a smaller string length than the foreign environment.

Output means data that is being written to the foreign environment. If there is a question of data loss during

output, the foreign environment is more limited than Advanced Revelation.

In addition to simply setting a degree of acceptable data loss, you can also specify values that are used for “overflow” (number too large) and “underflow” (number too small) conditions. For example, you might specify that if the bond encounters a floating-point number that is too large, that it substitute the overflow value 9999.999999.

Specifying Acceptable Data Loss

To set the degree of acceptable data loss for a particular field, follow these steps:

1. Use the options **Tools-Files-Dictionary** from the Advanced Revelation Main menu to display the Dictionary window.
2. Enter the name of the file and field that you are defining.
3. Press the [Shift-F5] Softkey to display additional field options.
4. At the *Acceptable Data Loss* prompt, enter a separate degree of acceptable data loss for both input and output. Use these values:

0	No data loss
1	Some data loss
2	Total data loss

5. If you have entered a value of 1 or 2 for acceptable data loss, you can enter additional information. Next to the degree values for input and output, enter overflow and underflow values for the field. These values are optional.
6. Save your changes using the [F9] Save key.

Null Handling

Some environments make a distinction between a field with no value and a field that is null. An integer field is a good example. If the field contains all zeroes, the value of the field is zero. However, if there is nothing whatsoever in the field, the field is null.

Although less common, this type of distinction is also sometimes found in character fields. In that case, there is a distinction between an “empty string” and a “null string”. In the former case, some designator in the field (perhaps a string of blanks) indicates that a string of no particular value has been assigned to the field. In the latter case, no value at all is assigned to the string.

- ✎ Advanced Revelation itself does not distinguish between an empty string and a null string. If Advanced Revelation encounters a string of length zero, it will treat it as a null string.

The fact that some environments distinguish nulls while others do not is a potential source for information loss. If you write a null field to an environment that has no concept of null, the “nullness” of the field is lost. Because Advanced Revelation does not distinguish null data from empty strings, the same problem occurs when you attempt to read null data into Advanced Revelation.

Accepting Null Data

By default, Environmental Bonding will not allow loss of information. It will therefore cause an operation to fail if the “nullness” of a field is lost in the process.

Two attributes in the dictionary definition for a field, however, permit you to specify how a bond is to handle nulls.

First, you can set a flag giving the bond permission to accept null data even if it has no way to indicate a null. In that case, the “nullness” of the field is lost. However, the bond will write an appropriate value back to the bonded file. For instance, the bond might write a string of blanks back to a character field to indicate a null.

Second, the bond may allow you to override the bond’s definition of “appropriate character” as a default null value. For example, you might want to have the bond write out the literal string “NULL” to a character field in order to indicate a null value. When the bond reads the record later, it will convert the string “NULL” into a true null value.

- ✎ Every bond establishes a default value for null for each data type. The documentation for a bond includes information about the default value for null.

To establish null handling for a field in your bond, follow these steps to allow null fields:

1. Use the options **Tools-Files-Dictionary** from the Advanced Revelation Main menu to display the Dictionary window.
2. Enter the name of the file and field that you are defining.
3. Press the [Shift-F5] Softkey to display additional field options.
4. At the *Accept Null* prompt, enter “Y”. If this prompt is blank, or if it contains “N”, the bond will not accept null data for this field.
5. At the *Null Char* prompt enter a value that you will recognize later as a null.



Indexing

The benefits of indexing in Advanced Revelation extend to bonded files as well. By establishing indexes for bonded files, you can gain performance increases in searching and sorting data.

Applying Advanced Revelation Indexing

Using an Advanced Revelation index with a bonded file is not generally recommended. Because Advanced Revelation indexes are only updated from within Advanced Revelation, other users of the foreign environment cannot keep Advanced Revelation indexes current.

The rule of thumb you should follow is this:

- If the bonded file will be updated from outside of Advanced Revelation, you should not use an Advanced Revelation index with the file.
- If the file will be updated *only* from within Advanced Revelation, then an Advanced Revelation index is recommended.



You may also want to consider avoiding the use of Advanced Revelation indexes if your bonded file type uses implicit keys. In that case, any operation that causes the implicit keys to change, such as “packing” the data, will invalidate the indexes. To keep the index current, you will need to rebuild it after every operation that changes record keys. However, if your file is stable (keys do not change), an Advanced Revelation index will be beneficial.

Foreign Indexes

Some foreign environments have their own indexing systems. For example, dBASE has a facility for creating, maintaining, and using indexes with dBASE files. If a bonded environment has its own indexing system, Advanced Revelation can take advantage of it when querying files. In fact, you can use Advanced Revelation and foreign indexes together in accessing your bonded files (subject to the recommendations above).

Use of Indexes with Bonded Files



Advanced Revelation will use both Advanced Revelation and foreign indexes when querying the file. Certain functions, however, such as direct index lookups in windows, will not use a foreign index. These are the rules for how Advanced Revelation will use indexes with bonded files:

- R/LIST and SQL will use any index (Advanced Revelation or foreign) for searching. R/LIST will also use any index for sorting. SQL will not use foreign indexes for sorting (ORDER BY), but will take advantage of them to optimize joins.

- In windows, any search method that generates an R/LIST sentence will use indexes as described above. This includes Query mode, EasyWriter, and [Ctrl-F10] filters.
- System functions that read indexes directly will use Advanced Revelation indexes, but not foreign indexes. This includes the system subroutines BTREE.INVERT, BTREE.EXTRACT, and COLLECT.IXVALS, as well as the codes and commands “B” and “V”.

Networks and Locking

If you are using an Environmental Bonding while on a LAN, you will need to know what kind of file and record locking facilities the bond supports. A bond might provide one of three levels of file and record locking:

- No locking at all.
- Locking coordinated with other Advanced Revelation users.
- Locking coordinated with users outside of Advanced Revelation.

No Locking

If a bond does not provide any locking at all, you will have to use alternative means to ensure that only one person updates a file and record at a time. One method is to set the network’s “share” flag for a file so that only one user can access it at a time. This method trades the convenience of multi-user access for the security of system-wide locks.

Locking within Advanced Revelation

One type of locking that a bond might support is the ability to set locks that are respected by other Advanced Revelation users. This is important if you anticipate that other Advanced Revelation users will be accessing the bonded files at the same time you are.

There is still the possibility of violating locks with users outside of Advanced Revelation. If you want to guard against this possibility, you will need to rely on a method such as that described above.

Locking in the Foreign Environment

A bond that supports the most complete locking protocol will coordinate all locks with users outside of Advanced Revelation. For example, the dBASE bond allows you to coordinate your locks with users who might be accessing the file from within the dBASE environment. If you lock records in a dBASE file, they will be respected by anyone else who might be accessing the file.

Chapter M 2

The ASCII Environmental Bond

This chapter explains the use of the ASCII Environmental Bond. With the ASCII bond, you can access existing ASCII files from within Advanced Revelation and create new ASCII files for use by Advanced Revelation and foreign applications.

Description of the ASCII Bond	M 2 . 1
Using the ASCII Bond	M 2 . 3



Description of the ASCII bond

The ASCII bond enables you to write or read an ASCII text file directly, without going through a time-consuming data import or export process. If you run applications that generate ASCII data files, you can use the ASCII bond to access these files within Advanced Revelation. You can also use the ASCII bond to create ASCII files that both Advanced Revelation and your foreign application can access and update.

The ASCII bond works with two types of ASCII text files: those that use delimiters to indicate variable field length, and those that have fixed field lengths. Once you indicate which file format you want to use and add field definitions to the data dictionary, the bond can access and maintain data in the file based on your specifications.

Variable vs. Fixed-length ASCII Files

Below are examples that show the same data formatted for a variable length file and a fixed length file.

Variable-Length ASCII File example

```
Smart-COMM,Delores Thompson,(408) 312-3533
Computer Micro-Systems,Scott Gerson,(704) 213-8732
Allied Software,John Draftson,(415) 555-1212
Alex Service Inc,Alex Southward,(801) 853-0987
ACM Accounting,Allen Jacobson,(612) 497-2364
```

In a variable-length file, the data written to a field can be any length. The end of each field is indicated by a field delimiter. In the example above, the delimiter is a comma. Fields are located by their position in the record and each field is separated by a comma.

Fixed-Length ASCII File Example:

Smart-COMM	Delores Thompson	(408)	312-3533
Computer Micro-Syste	Scott Gerson	(704)	213-8732
Allied Software	John Draftson	(415)	555-1212
Alex Service Inc	Alex Southward	(801)	853-0987
ACM Accounting	Allen Jacobson	(612)	497-2364

In a fixed-length file, each field in the record has a predetermined length, which is the amount of space allocated for that field. If you write less than the maximum number of characters to a fixed-length field, the data is padded with spaces so that the next field in the record begins in its proper position. You cannot write more than the number of characters allocated for that field. For example, in the second record shown above, only part of company name — “Computer Micro-Syste” — could fit in the first field. If you look at the second record in the variable-length file example, you see that the length of the data stored in the field is dynamic rather than fixed, so the entire company name can fit into the field.

Installation

You can install the ASCII bond using the installation procedure for optional modules. See the “Updating and Enhancing Your Advanced Revelation System” chapter in the *Installation* section for details on this process.

Characteristics of the ASCII Bond

The structure of ASCII files requires that you keep a few issues in mind when using the ASCII bond. Details about the characteristics listed below appear in the “Using the ASCII Bond” section.

Flat-file Architecture	Because ASCII files use a flat-file structure, both records and fields have absolute locations corresponding to their order in the file. It is your responsibility to create the proper sequence of records and fields.
ASCII File Types	The ASCII bond enables you to create and access fixed-length and variable-length ASCII text files.
Accessing Binary Data	You cannot use the ASCII bond to access ASCII files that contain data in binary format.
Illegal Delimiter Characters	The ASCII bond reserves several characters for use exclusively by the bond. You should not use any of the following characters for record or field delimiters or for the text marker: ASCII CHAR 19 ASCII CHAR 20 ASCII CHAR 21 ASCII CHAR 252 (subvalue mark) ASCII CHAR 253 (value mark) ASCII CHAR 254 (field mark)
Implicit Keys	The ASCII bond uses “implicit” keys to identify records. The keys are implicit because they do not exist as data in the ASCII file. Instead, the record keys indicate the position of each record in the file, beginning with record key 1 for the first record and continuing sequentially.
Indexing	You should apply Advanced Revelation indexing (SI.MFS) <i>only</i> to ASCII files that will not be accessed or updated by processes outside of the Advanced Revelation environment.
Maximum Record Length	The ASCII bond can work with an ASCII file of any size. However, individual records in the file may not be longer than 10K.

Locking The ASCII bond does not support record or file locking. ASCII files can be corrupted if two or more users attempt to write to the same file at the same time. Advanced Revelation users should not access ASCII files that are being used by other users in a multiuser or network environment.

Null Handling The ASCII bond makes no distinction in string fields between a null value and an empty string.

Using the ASCII Bond

Working with a bonded ASCII file is very similar to using any other Advanced Revelation file. You can use most of the same tools and perform most standard operations on ASCII files.

- ☛ The “Introduction to Bonding” chapter in *Environmental Bonding* provides background information about standard Environmental Bond behavior.

Any requirements or distinctive operations involved in using the ASCII bond are explained below. In all other circumstances, you should assume a bonded ASCII file requires no special treatment.

Accessing ASCII Files

You must create a volume for your ASCII files before using them. A volume will tell Advanced Revelation that you are using the ASCII “file type”, plus the location of your ASCII files. You can create multiple volumes if you have ASCII files in more than one location.

To assign a volume, choose the **Tools-Files-Set Volume** options from the Advanced Revelation Main menu or use the SETVOLUME command from TCL.

Fill in the prompts as for any volume (for information on volumes, see the chapters “Advanced Revelation Volumes” in *Filing Systems* and “Introduction to Bonding”). At the *File Type* prompt specify “ASCII_BFS” to indicate that the files in this volume are ASCII files. If you do not fill in a value at the *Control Files Location* prompt, the ASCII bond will place the control files in a default location (the sub-directory ASCII beneath the current subdirectory).

The files in an ASCII volume must be attached in order to make them available to Advanced Revelation processes. Choose the **Tools-Files-Attach** options from the Advanced Revelation Main menu, or use the ATTACH command from TCL.

Using Existing ASCII Files

ASCII files typically do not store any information about their file structure. Therefore, when you first access an existing ASCII file, Advanced Revelation creates only a rudimentary data dictionary for the file. To successfully use the ASCII bond, you

must expand on the initial data dictionary information — providing field definitions and specifying appropriate file attributes. *This is not an automatic process.* If you do not supply this dictionary information, the ASCII bond cannot read or write to an ASCII file.

The dictionary that Advanced Revelation creates for an ASCII file is assigned the standard Advanced Revelation identifier of:

`DICT.filename`

where *filename* is the name of the ASCII file. To display the dictionary, use the TCL command LISTDICT.

The initial data dictionary for an ASCII file defines only one field — the implicit key field. You must make these changes to the file's data dictionary so that it accurately describes the file and record structure:

- Specify appropriate file attributes.
- Add field definitions to describe the record structure.

Warning! Once you complete the file definition process, you should test the accuracy of your current file definition by querying the file. *Do not attempt to add or delete data in an existing ASCII file unless you are confident that the current dictionary is accurate.* For more information, see “Testing the File Definition” later in this chapter.

The Definefile window enables you to define the file attributes and field characteristics of an existing ASCII file. To access this window, choose the **Tools-Files-Define File** options from the Advanced Revelation Main menu.

- ☞ In addition to the information provided below, you can find documentation about the use of the Definefile window in the “Managing Files and Records” chapter in *Filing Systems*.

Specifying ASCII File Attributes

When you access an existing ASCII file, the ASCII bond assumes that the file structure complies with a set of default file attributes:

- The file is variable-length
- The field delimiter is a comma
- The record delimiter is carriage return/linefeed

If the default attributes are not correct for a given file, various problems may arise. For example, if the file contains no carriage return/linefeed sequences, the file will appear to be one big record. In that case, if the file is bigger than the maximum record size, attempts to access the file will result in an error.

In order to read and write the records in an ASCII file properly, appropriate file attributes *must* be specified. You must

inform the ASCII bond if your ASCII file is a fixed-length file or if your delimiters differ from the standard variable-length file delimiters.

- ✎ If a foreign application generated your ASCII files or will be accessing them, see the documentation for the foreign software to find out which file format and delimiters to use.

Changing File Attributes

Make changes to the current file attribute specifications at the Definefile window. To do so:

1. At the *Filename* prompt, enter the name of the ASCII file that you want to work with.
2. At the *Volume* prompt, enter the volume assigned for the file. The current dictionary information for the file will display.
3. Press [Shift-F1] to display the ASCII File Attribute window. You can press [F1] for online help at any of the prompts in the window.
4. Enter the correct file attributes at the prompts in the window and press [F9] to save your specifications.

For more information about each of the prompts in the ASCII File Attribute window, see “Defining Files” later in this chapter.

Defining ASCII Fields

To create a new ASCII file or access an existing one, you must define all of the data fields in the record. The ASCII bond cannot recognize the individual fields until you add the appropriate field definitions to the data dictionary.

- ✎ If a foreign application generated your ASCII files or will be accessing them, see the documentation for the foreign software to find out about the record structure and field characteristics of the file.

If you intend to make updates to a bonded ASCII file, the Advanced Revelation dictionary must accurately describe the existing file structure.

Use the lower half of the Definefile window to enter field definitions. For more information, see the “Managing Files and Records” chapter in *Filing Systems*.

Data Types

The ASCII bond supports the bond-specific ASCII_STRING data type and all of the standard Advanced Revelation data types except TEXT. If the type you specify requires additional parameters, a collector window will display. Fill in the

prompts in the collector window and press [F9] to save your specification.

ASCII_STRING

There is one data type, `ASCII_STRING`, that is unique to the ASCII bond. Unlike the standard data types, the `ASCII_STRING` data type has no default characteristics.

When you use the `ASCII_STRING` data type, a collector window displays. You must specify two parameters: justification and length. Optionally, you can specify conversions for data input and output.

- ✎ Your `ASCII_STRING` parameter specifications for a field are also used as that field's default foreign field attributes.
- *Input Conversion* specifies the internal format that Advanced Revelation should convert data to when it is read from the ASCII file. See `ICONV` in the “R/BASIC Command Reference” chapter in *R/BASIC* for a list of valid conversion specifications.
- *Output Conversion* specifies the format that Advanced Revelation should convert the data to when it is written to the ASCII file. See `OCONV` in the “R/BASIC Command Reference” chapter in *R/BASIC* for a list of valid conversion specifications.
- At the *Justification* prompt you can specify one of three ways that the data will be justified — L (left), R (right), or C (centered). The *Justification* specification is ignored for variable-length fields.
- If you are defining a fixed-length field, the *Length* parameter determines the maximum number of characters that can be stored in the field and the default display length for the field.

If you are defining a variable-length field, *Length* does not limit the number of characters that can be entered in the field. Instead, it serves as a default data display specification, determining how many characters display on reports.

Specifying Foreign Field Attributes

You can specify foreign field attributes for each field that reads and writes data to the file. The foreign field attributes define precisely how the data should be read for use in Advanced Revelation, and how it should be written to the ASCII file.

- ☞ All data types include a set of default foreign field attributes. You only need to specify foreign field attributes if you do not want to use the defaults.

To view or change the current foreign attributes for a field, press [Shift-F3] at the Definefile window. Then enter the name of the field at the prompt. A collector window displays so that you can view and enter attribute information.

The foreign field attributes entry window and the ASCII_STRING data type parameter entry window both prompt for similar information. See “ASCII_STRING” above for details about what to enter at the *Input*, *Output*, and *Justification* prompts.

- If you are defining a fixed-length field, *Length* specifies the maximum number of characters that can be stored in the field. If you write less than the maximum number of characters to the field, the data is padded with spaces to equal your specification.

For variable-length fields, you can use the *Length* specification to determine the minimum number of characters that can be stored in the field. If you write less than the minimum number of characters to the fields, the data is padded with spaces to equal your specification. If you write more than the minimum number of characters to the field, the *Length* specification is ignored.

- If you are defining a variable-length field, you can enter Y at *Use text marker (Y/N)?* to specify that data should be surrounded by the text marker when it is written to the file. The character used as a text marker is specified as part of the file attributes. A " (quote) character is the default text marker.

- ☞ For more information about text markers, see “Specifying ASCII File Attributes” later in this chapter.

Testing the File Definition

To determine whether or not your current file definition is accurate, use R/LIST to view the contents of several records in the file. To do so:

1. Press [F5] to display the TCL window.
2. Enter a LIST command, specifying how many records to list and naming all of the fields in the record. The command syntax is:

```
LIST {n} filename fieldname(s)
```

For example, this command lists 5 records in a CONTACTS file that has fields for name, city, state, and phone number:

```
LIST 5 CONTACTS NAME CITY STATE PHONE
```

The report will display in the View window. Each column following the key represents a field in the file. Just by looking at the data displayed in the report columns, you should be able to determine whether your current definition of the file structure is accurate.

If data appears in the wrong columns:

- Fixed-length file —
Did you specify the field lengths incorrectly? Is your record delimiter correct? Did you define all of the fields in the record?
- Variable-length file —
Are you using the proper file attributes for record and field delimiters? Are you using a text marker correctly? Did you define all of the fields in the record?

Defining Files

To create a new ASCII file, choose the **Tools-Files-Define File** options from the Advanced Revelation Main menu, or use the DEFINEFILE command from TCL. Refer to the “TCL Command Reference” chapter in *TCL*. In addition to the information provided here, you can find documentation about the use of the Definefile window in the “Managing Files and Records” chapter in *Filing Systems*.

File Names Although the Advanced Revelation name for your bonded file can be any valid Advanced Revelation identifier, this name may not be a valid file name at the operating system level.

If you are creating an ASCII file and your filename specification is illegal, the ASCII bond will display a message. The message indicates that the bond has assigned an alternate *foreign* filename for use outside Advanced Revelation: *filename.DAT*.

Press [Enter] to accept the assigned foreign name or enter the foreign name you prefer. The file name that you enter must use the standard DOS file naming conventions:

- a filename of from one to eight characters
- a dot (“.”)
- a filename extension consisting of three or fewer characters

If you do not specify a filename extension when creating a new file, the bond automatically assigns the extension “.DAT” to the filename.

Specifying ASCII File Attributes

Use the *File Attributes* prompt in the Definefile window to specify the correct file attributes.

Press [F2] at the *File Attributes* prompt to display the ASCII File Attribute window. Attributes are entered at the following prompts:

File Type

You must specify whether the ASCII file is fixed-length or variable-length. Enter F for fixed, V for variable. The default is V.

☛ If you specify an F (fixed-length) file type, the prompts for *Field Delim*, *Value Delim*, and *Text Marker* are removed from the window.

Header Cnt

To define a file header at the beginning of the ASCII file, enter the number of bytes to reserve for the header. A specification at this prompt tells the ASCII bond not to write to this portion of the file. Only data that is stored after the header is accessible by the ASCII bond.

Record Delim

The record delimiter is required when you are defining an ASCII file that uses variable field length record format. It specifies the ASCII character used in the file to indicate the end of a record. The record delimiter may be 1 or 2 bytes in length. The default delimiter is a carriage return and linefeed (13:10).

Enter the ASCII decimal number of the record delimiter or press [F2] to display the ASCII Character Set popup, which lists all ASCII characters.

If the record delimiter is 2 bytes, separate the two ASCII characters with a : (colon). For example, you might enter:

13:10

if the record delimiter in the ASCII file is a carriage return and linefeed.

Field Delim

The field delimiter is required when you are defining an ASCII file that uses a variable-length fields. The field delimiter specifies the ASCII character that indicates the end of a field. The field delimiter is limited to 1 byte in length. The default delimiter is a comma.

Enter the ASCII decimal number of the field delimiter or press [F2] to display all ASCII characters. For example, you might enter:

44

if the field delimiter in the ASCII file is a comma.

Value Delim A value delimiter specifies the ASCII character used to separate the individual values in a multivalued field. The value delimiter is limited to 1 byte in length.

Enter the ASCII decimal number of the value delimiter or press [F2] to display all ASCII characters.

Text Marker A text marker is used to distinguish between a character that is actually data in a field, and a character that is being used as the field delimiter. For example, an ASCII file that uses commas as field delimiters may contain a text field that includes commas. Specifying a text marker ensures that the bond recognizes commas in data as text rather than as delimiters.

A text marker specifies that when data in a field contains the character that is specified as the field delimiter, that data is enclosed by the text marker. The text marker is limited to 1 byte in length. In most cases, the text marker is " (a double quote mark).

Some products create ASCII files in which all fields containing non-numeric data are enclosed in a text marker character. If you are writing to such a file, you can specify that data in designated fields is *always* surrounded by the text marker when it is written to the file. This specification is made on a field-by-field basis, using the softkey [Shift-F3] (Specify Foreign Field Attributes). For more information, see "Specifying Foreign Field Attributes" under the topic "Defining ASCII Fields" earlier in this chapter.

Enter the ASCII decimal number of the text marker or press [F2] to see a list of all ASCII characters.

Assigning Keys The ASCII bond automatically creates an implicit record key field at the time that the dictionary file is created. The implicit key field in the dictionary is called RECNO and its data type is INTEGER.

 You cannot select any other field to be the key to an ASCII file. You cannot create a multipart key for an ASCII file.

Synonyms for RECNO When you use Paint to define an ASCII file, the prompt that you identify as the key becomes a synonym for the implicit key RECNO and is added to the dictionary.

You can also add a synonym for RECNO at the Dictionary window or at the Definefile window.

Defining Fields To create a new ASCII file or access an existing one, you must define all of the data fields in the record structure.

Use the lower half of the Definefile window to enter field definitions. For more information, see "Using Existing ASCII Files" above. See also the "Managing Files and Records" chapter in *Filing Systems*.

Field Position Gaps Every "F" type field is assigned a position in the record. When you define an ASCII file, field position gaps are permitted. For example, you might have fields 1 through 3, skip position 4, and then continue with fields 5 through 10.

☞ You can display field position information in the Definefile window, the Dictionary window, or using the TCL command LISTDICT.

Copying Files and Records

You can use the usual commands in Advanced Revelation to copy and move data in the ASCII bond. However, there are restrictions on doing so.

Copying Records Unless the record structure of both the source and destination file is exactly the same, you should not copy data using the TCL command RECORDCOPY. Any differences in field definitions will prohibit the copy operation or may result in data loss.

Copying Files To copy an entire ASCII file, use the FILECOPY command. As with other file types, you can only copy files within a single environment. This means that you can copy an ASCII file to another ASCII file only.

Modifying Files

Changes you make to bonded ASCII files may be affected by the constraints imposed by the ASCII file structure. Both record length and order are important.

☞ You cannot use the TCL command REMAKEFILE to convert an ASCII file to another file type.

Warning ! If you intend to make updates to a bonded ASCII file, the Advanced Revelation dictionary must accurately describe the existing file structure. If the dictionary is not accurate, you run the risk of overwriting and corrupting existing data in the bonded ASCII file when records are added or deleted.

Writing to Fields If you inadvertently violate a field definition by writing invalid data to it, one of several results may occur. The results depend on how you have set the degree of acceptable data loss for the field.

☞ For more information about potential data loss, see the chapter “Introduction to Bonding.”

If you have set the degree of acceptable data loss to zero (no loss allowed), the write operation will fail. You may see a message informing you of this. If the degree of loss is set to 1 or 2, the write operation may succeed, but you may experience partial or total data loss.

Adding Records You can add records to a bonded ASCII file at any time. However, if you add a record out of sequence, past the end of the file, empty records (fixed-length files) or record delimiters (variable-length files) are appended to the file. This is necessary in order to assign implicit keys correctly, since the keys are derived from the actual order of the records in the file.

In a fixed-length ASCII file, for example, if there are 10 records in the file and you add record 100, ninety records are appended to the file — 89 empty records padded with spaces, followed by the data record 100.

In a variable-length ASCII file, if there are 10 records in the file and you add record 100, 89 record delimiters are appended to the file, followed by the data record 100.

For this reason, it is a good idea to assign the sequential counter %SK% to key prompts when creating windows that access and update ASCII files. For more information about the sequential counter, see the chapter “Adding Advanced Features to Windows with Paint” in *Developing an Application*.

Deleting Records Records are deleted from ASCII files by using the DELETERECORD command.

Deleting Variable-Length Records

When you delete a record from a variable-length ASCII file, only the data is removed. The record delimiter is left intact. This ensures that the position of the remaining records is not changed and that the implicit record keys (location in the file) remain the same.

Deleting Fixed-Length Records

When you delete a record from a fixed-length ASCII file, the data in the record is converted to spaces. This is done to preserve all record positions in the file and maintain the implicit keys. If you attempt to read the deleted record back, the read operation will fail. See "Null Handling" below.

Null Handling

In ASCII files, there is no explicit way to indicate a null record. Instead, an empty string or a string of spaces is used to designate a null record or field.

The ASCII bond makes no distinction in character fields between a null value and an empty string. In variable-length files, the value for NULL is an empty string. In fixed-length files, the value for NULL is blanks (spaces).

In a bonded ASCII file, a record of all null fields (no data whatsoever) is treated as though it does not exist. You cannot deliberately save a null record to the file and then read it back.

For instance, suppose you attempt to write out 5 blank spaces to a fixed-length field that is 5 characters long. When you read the field back in, it will come back as null, because the ASCII bond is incapable of distinguishing between blank spaces and null values.

Indexing

You should apply Advanced Revelation indexing (SI.MFS) *only* to ASCII files that will not be accessed or updated by processes outside of the Advanced Revelation environment.

The position of records in the ASCII file may shift when a foreign process adds or deletes records. Since the implicit key is derived from the location of the record, a change in position would change the key assigned to the record.

Anything that causes the implicit keys to change will invalidate the indexes. To keep the index current, you will need to rebuild it after every operation that changes record keys.

Networks

Currently, the ASCII bond supports single user access only. The ASCII bond does no record or file locking. Because no record or file locking is performed, ASCII files can be corrupted if two or more users attempt to write to the same file at the same time. For this reason, you should not allow Advanced Revelation users to access ASCII files that are maintained in a multiuser or network environment.

ASCII files that are accessed with the ASCII bond in Advanced Revelation should be isolated in a single user environment. For example, they might be stored on your local hard disk. Or, your network operating system may provide a method to make files non-shareable.

Default Foreign Field Attributes

The following table lists the default ASCII foreign field attributes for each data type. These attributes determine precisely how data is read for use in Advanced Revelation, and written to the ASCII file. The Justification and Length defaults in the table are for fixed-length ASCII files. For variable-length ASCII files, Justification and Length default to null.

☞ If you do specify Length for a variable-length ASCII file, ASCII will pad the field to the appropriate length. When Length is specified, the default Justification is L.

ASCII Foreign Field Attributes

Type	Output	Input	Justification	Length
CHAR			L	required
VARCHAR			L	256
INTEGER	MD0	MD0	R	4
DECIMAL	MD0,	MD0	R	1
FLOAT			R	20
DOLLARS	MD2,\$	MD2	R	14
BOOLEAN	BY,N	BY,N	L	1
DATE	D4/	D	R	10
TIME	MTHS	MT	R	10
DATETIME	DTD4/^HS	DT	L	21

Chapter M 3

The dBASE III Environmental Bond

The dBASE III Environmental Bond allows you to access data stored in the format of used by dBASE III. Using this bond, you can read and write data in a dBASE file as if it were in an Advanced Revelation file. This chapter describes how to use the dBASE III Environmental Bond.

Description of the dBASE III Bond	M 3 . 1
Using the dBASE III Bond	M 3 . 3
The dBASE III Bond Reference	M 3 . 12



Description of the dBASE III Bond

The dBASE III environmental bond enables you to use Advanced Revelation as a front-end to your dBASE III or dBASE III Plus environment. This means that new and existing Advanced Revelation applications can access from dBASE files. Using the dBASE bond, you can even create new files that will be accessible to dBASE users.

When you use the dBASE bond, your dBASE files are maintained in their proper format. Using Advanced Revelation, you can still perform the normal dBASE operations of creating, modifying, or deleting dBASE files or records. At the same time, you have the added sophistication provided by Advanced Revelation tools such as Paint and R/BASIC. Changes you make from within Advanced Revelation are reflected for users who access the data from within dBASE.

- ☛ For more information about Environmental Bonding, see the chapter "Introduction to Environmental Bonding".

Installation and Logon Options

You can install the dBASE bond using the installation procedure for optional modules. See the *Installation* section of the documentation for details on this process.

Logon Options When you open dBASE data files in Advanced Revelation, there is a potential to exceed the number of file handles that are normally available. This is because all dBASE indexes and Memo fields are maintained as separate files. Since Advanced Revelation enables you to have more than one dBASE file open at a time, the available file handles can be exhausted rapidly.

To eliminate this problem, Advanced Revelation provides an /H (Handles) logon option. The H option enables you to increase the number of file handles that are reserved for use by Advanced Revelation.

- ☛ The /H option is only supported on systems using DOS version 3.3 or greater.

Increasing the Available File Handles

By including the H option in your logon command, you can increase the number of file handles available to Advanced Revelation. For example, the SYSPROG user might enter:

```
AREV SYSPROG /Hnnn
```

where the slash (/) indicates that multiple logon options may follow, and *nnn* is a number from 20 to 255 specifying the number of file handles to reserve.

The number of file handles reserved by Advanced Revelation can never exceed the FILES specification in the current CONFIG.SYS file.

- ☛ To ensure that sufficient file handles are available to Advanced Revelation, the FILES specification in your CONFIG.SYS file should be set at 30 or more.

The dBASE III Bond Environment

The dBASE bond provides an additional menu and window and an additional TCL command as extensions to the standard Advanced Revelation environment.

- | | |
|----------------------------|---|
| The dBASE III Menu | To display the dBASE III Filing Systems menu, choose the Tools-Bonds options from the Advanced Revelation Main menu. |
| The Indexing Window | To display the Indexing window, choose the single option Indexing at the dBASE III menu. |
| The DBASE Command | You can use the DBASE command to display either the menu or window listed above or to associate existing dBASE indexes with a file. |

- ☛ See “Indexing” below for more information.

Characteristics of the dBASE III Bond

The constraints of the dBASE architecture require that you keep a few issues in mind when using the dBASE bond:

- | | |
|-------------------------------|---|
| Flat-file Architecture | Because dBASE is a flat-file database, both records and fields have absolute locations corresponding to their order in the file. It is your responsibility to create the proper sequence of records and fields. |
| Fixed-length Records | The dBASE files are based on a fixed-length record structure. Consequently, the dBASE bond will not allow you to make any changes to a file that might require structural changes. If you need to make such changes, you must make them within the dBASE environment. |

Implicit Keys	The dBASE bond uses “implicit” keys to identify records. The keys are implicit because they do not exist in the dBASE file but are instead derived from the location of the record in the file. Implicit key assignments are necessary for the dBASE bond to ensure compliance with dBASE architecture. The dBASE bond maintains the proper record order using sequential, implicit keys for the records.
Maximum Number of Records	The dBASE environment does not allow the creation of more than one billion records.
Indexing	You can create and use dBASE indexes for a bonded dBASE file. You can also apply Advanced Revelation indexing (SI.MFS) to a bonded dBASE file, although this is not recommended if the file will be accessed from outside of Advanced Revelation.

Using the dBASE III Bond

A bonded dBASE file is for the most part no different than any other Advanced Revelation file. You can use all of the same tools and perform all of the standard operations on dBASE files with the exception of indexing.

Special requirements or operations required to use the dBASE bond are discussed below. In all other circumstances, you should assume a bonded dBASE file requires no special treatment.

Accessing dBASE Files

You must create a volume for your dBASE files before using them. A volume will tell Advanced Revelation that you are using the dBASE “file type,” plus the location of your dBASE files. You can create multiple volumes if you have dBASE files in more than one location.

To assign a volume, choose the **Tools-Files-Set Volume** options from the Advanced Revelation Main menu or use the SETVOLUME command from TCL.

Fill in the prompts as for any volume (for information on volumes, see the chapters “Advanced Revelation Volumes” in *Filing Systems*, and “Introduction to Environmental Bonding”). When indicating a *File Type* for the dBASE bond, use the name DBASE_BFS. If you do not fill in a value at the *Control Files Location* prompt, the dBASE bond will place the control files in a default location (the subdirectory DBASE beneath the current subdirectory).

After you have assigned a volume, you need to attach the volume to make it accessible. Choose the **Tools-Files-Attach** options from the Advanced Revelation Main menu, or use the ATTACH command at TCL.

Using Existing dBASE Files

You can access existing dBASE files by simply attaching them (after creating a volume). When you first access an existing dBASE file, an Advanced Revelation dictionary is dynamically created for the file. The existing fields in the dBASE file are mapped to corresponding dictionary definitions in Advanced Revelation. In addition, an implicit key field is created in the Advanced Revelation dictionary.

The dictionary created for a file is assigned the standard Advanced Revelation identifier of:

`DICT.filename`

where *filename* is the name of the existing dBASE file. To display the dictionary, use the TCL command LISTDICT.

Modifying the Dictionary

Except in the case of assigning a synonym for a field (see “Assigning Keys” below), adding or deleting a dictionary definition for an existing data file would require a structural change to the file. Currently, you cannot do this unless you first delete the file.

☛ See “Modifying Files” below.

You can add or delete dictionary definitions as long as the data file does not exist.

When using Paint, for example, provided that you do not choose to create the data file, you can add or delete field definitions at will.

Once the file has been created, however, you can only change the dBASE file by either deleting and redefining it, or by restructuring it from within the dBASE environment.

Defining New dBASE Files

To create a new dBASE file, choose the **Tools-Files-Define File** options from the Advanced Revelation Main menu or use the DEFINEFILE command at TCL. For more information on the Definefile window, see the chapter “Managing Files and Records” in the *Filing Systems* section.

Assigning Keys

When you define a dBASE file, an implicit key field called RECNO is automatically created in the dictionary. At the Paint window, the name you choose as the key field for your file becomes a synonym for the RECNO field and is added to the dictionary.

You can also add a RECNO synonym at the Dictionary window.

Because dBASE uses implicit keys, there is the possibility that the primary key to a record can change. For instance, if a user “packs” a file from within the dBASE environment in order to compress out deleted records, the remaining records can have new keys (positions) assigned to them. The net effect from within Advanced Revelation is that the key field for a record will have changed.

Data Typing

There are a few data types provided that are unique to the dBASE bond. These dBASE types reflect their usage in the dBASE environment. You can use these types or the standard Advanced Revelation data types.

The dBASE bond-specific types are:

- DBASE_C (character)
- DBASE_D (date)
- DBASE_N (numeric)
- DBASE_L (logical)
- DBASE_M (memo)

If you use the standard Advanced Revelation data types, correspondences will be automatically applied to map to an appropriate type for the dBASE bonding dictionary.

☞ See *Tables M3-1, 2, and 3* in “The dBASE III Bond Reference” below for bonding dictionary information and the standard type correspondences.

The following are the types provided by the dBASE bond:

- DBASE_C** This type of field may be any printable ASCII character. The maximum length is 254 characters.
- DBASE_D** The DBASE_D field is an internal representation of the date as maintained by the dBASE environment.

DBASE_N A DBASE_N field may be an integer or decimal number. The maximum length is 19 digits, where a decimal point and negative sign each count as a digit. The maximum number of decimal places is 15.

DBASE_L Fields of this type indicate logical true or false. The maximum length is one character. The internal representation as maintained by the dBASE environment is 0 or 1.

DBASE_M A DBASE_M field may be any printable ASCII character. The maximum length is 5,000 characters of text.

Assigning Field Attributes

Whenever you define a dBASE field, only two attributes are potentially available: length and decimal places. Their applicability is determined by the data type you choose.

DEFINEFILE will prompt you for the attributes when you enter the data type.

Maintaining Field Position

Because fields have a fixed location in dBASE record structures, you must maintain an ordered sequence of fields in the dictionary. In other words, there must be a field defined for every position in the record.

- ☛ Each field has a *Position* number. You can display this information in the Definefile window, in the Dictionary window, or by using the TCL command LISTDICT.

The bond will not allow you to create a file with missing fields. If you delete a field when defining the dictionary (before the data file exists), you will need to change the position numbers assigned to the remaining fields to reflect the new sequence.

Updating dBASE Files

Using your dBASE bond, you can update dBASE files as if they were Advanced Revelation files. At the same time, users of the dBASE environment will see your changes as if they had been made in the dBASE environment.

The following topics address issues of which you should be aware as you update dBASE files.

Writing to Fields

Any data you write to a field must conform to the type, length, and possibly decimal places of the field as defined in the dictionary.

If you inadvertently violate a field definition by writing invalid data to it, one of several results may occur. These depend on how you have set the degree of acceptable data loss for the field.

- ☛ For more information about potential data loss, see the chapter "Introduction to Environmental Bonding".

If you have set the degree of acceptable data loss to zero (no loss allowed), the write operation will fail. You may see a message informing you of this. If the degree of loss is set to 1 or 2, the write operation may succeed (it will not be rejected), but you may experience partial or total data loss.

Adding a Record

You can add records to a bonded dBASE file at any time. However, if you add a record out of sequence, past the end of the file, records marked as "deleted" are appended to the file.

- ☛ See "Deleting a Record" below.

For example, if there are 10 records in a file and you add record number 100, 89 "deleted" records are appended to the file. For this reason, it is a good idea to assign the sequential counter %SK% to key prompts when creating dBASE records.

- ☛ For more information about the sequential counter, see the chapter "Adding Advanced Features to Windows with Paint" in *Developing an Application*.

Deleting a Record

When you delete a record, the data is removed, but the position in the file for the record is maintained. In order to preserve the dBASE architecture, the record is marked as deleted and its key (location in the file) remains.

If you subsequently write to the deleted record, it is automatically marked as undeleted and the new contents are added.

Copying Files and Records

You can use the usual commands in Advanced Revelation to copy and move data in the dBASE bond. However, there are restrictions on doing so.

Copying Records

In many cases, you can use the RECORDCOPY command to copy individual records to and from a dBASE file. As a rule, the target environment determines whether you can copy a record to it. In the case of the dBASE bond, the records you are copying to a dBASE file must match the file structure. In other words, each field of each record must match the dBASE definition for that field (data type, length, etc.). If you

attempt to copy a record with invalid data to a dBASE file, the copy operation will fail.

- ☛ If you have set the degree of acceptable data loss for a field to 1 (accept some data loss) or 2 (accept total data loss), the copy operation might be successful even if there is a mismatch. However, you may end up losing some data. For more information about potential data loss, see the chapter “Introduction to Environmental Bonding”.

On the other hand, it is relatively easy to copy records from a dBASE file into an Advanced Revelation file, since an Advanced Revelation file supports a much more flexible record structure.

Copying Files To copy an entire dBASE file, use the FILECOPY command. As with other file types, you can only copy files within a single environment. This means that you can copy a dBASE file to another dBASE file only.

- ☛ You may find it easier to copy the dBASE file using an operating system COPY command. You can then attach the copied file from within Advanced Revelation, and the dictionary for it will be created automatically.

Modifying the File Structure

Changes you make to bonded dBASE files may be affected by the constraints imposed by the dBASE file structure. Both record length and order are important.

Rebuilding a File When you need to rebuild a file, the issue is whether:

- The file should reflect the dictionary definitions.
- The dictionary definition should reflect the file structure.

Modifying the File Structure

Currently, you cannot alter the structure of a dBASE file from within Advanced Revelation. However, you can change the structure by deleting the file and rebuilding it. To do this, use the DELETEFILE command. Refer to the “TCL Command Reference” chapter of *TCL*.

Warning ! Deleting the file removes all data from it *permanently*. If you need to save the data, copy it first to another file, then restore it. You may need to use an R/BASIC program to do this.

- ☞ You cannot use the TCL command REMAKEFILE.

Although you cannot change the file structure from within Advanced Revelation, you can do so from within the dBASE environment. In fact, such a change may occur without your knowledge if another user decides to change the file.

When the structure of a dBASE file changes, you must recreate the Advanced Revelation dictionary for the file. To do so, simply delete the dictionary for the file. When you next access the file, a new dictionary will be automatically created based on the existing file structure.

- ☞ If you have created symbolic fields or synonyms, you will want to copy these to a temporary file before deleting the dictionary. After the new dictionary has been created, copy these fields back from the temporary file.

To delete the dictionary of the file, use the DELETEFILE command *including* the designation "DICT." in front of the file name. An example illustrates this:

```
DELETEFILE DICT.CUSTOMER
```

Warning! If you do not include the designation "DICT.", the *data* portion of the file will be deleted.

Modifying the Dictionary

To make any changes to the dictionary you must first delete the data file, otherwise the bond will not allow the changes. The appropriate steps are:

1. Delete the data file.
2. If you want to modify the existing dictionary, make these changes, being certain that the dictionary name reflects the intended data file name.
3. Define and create a new data file using the Definefile window or the TCL command DEFINEFILE.

Indexing

You can use dBASE indexes with your bonded dBASE files. You can also add Advanced Revelation indexes to your bonded dBASE files. This is *not* recommended if there is any chance whatsoever that users will add or delete records to the file from outside of Advanced Revelation, or if dBASE users might "pack" the file to compress out deleted records. If any of these operations occurs outside of Advanced Revelation, the Advanced Revelation index will be out of date, and will have to be rebuilt.

- ☞ There is no automatic way from within Advanced Revelation to detect whether an index will need to be rebuilt.

When you use a dBASE index using the dBASE bond, you have the following capabilities:

- Associate existing dBASE indexes with a file.
- Create new dBASE indexes.
- Redefine dBASE indexes.
- Disassociate dBASE indexes from a file.

You can associate up to nine indexes with a bonded file. When you associate indexes with a file, the dBASE bond maintains the association between your sessions — you do not need to re-associate the indexes the next time you log into Advanced Revelation.

Both R/LIST and SQL will automatically use any dBASE indexes associated with a bonded file for reduction, as long as the indexed expression is a field name.

Associating Existing Indexes

In order to associate existing indexes with a bonded file, both the file and the indexes must exist on the same volume.

There are two methods you can use to associate existing dBASE indexes with a bonded dBASE file:

The Indexing Window

Access the dBASE Indexing window by choosing the **Tools-Bonds-dBASE-Indexing** options from the Advanced Revelation Main menu.

The Indexing window consists of three prompts. At each prompt you can press [F2] to display a listing of optional entries for the prompt, or [F1] for online help.

To associate an index, enter:

1. The name of the bonded file at the *File Name* prompt. Do not use a file extension — the extension .DBF is assumed. After entering the file name, any indexes currently associated with the file will be displayed.
2. The name(s) of the dBASE index(es) you wish to associate at the *Index Name* prompt. You can specify up to nine indexes. Do not use a file extension — the extension .NDX is assumed.
3. The existing field name or expression indexed at the *Index Expression* prompt

- ☞ Although the dBASE bond will maintain existing complex expressions (such as equations and functions) used at the *Index Expression* prompt, you will not be able to create new indexes using complex expressions.

Press [F9] to associate the index(es).

The dBASE Command

The dBASE command enables you to associate existing indexes from TCL. However, you cannot create new dBASE indexes using the dBASE command.

The syntax for the dBASE command is:

```
DBASE SET INDEX bonded_file TO index
```

where *bonded_file* is the name of the bonded dBASE file, and *index* is the name of the index(es) to associate. The file extensions .DBF and .NDX are assumed, and should not be specified.

One form of the dBASE command displays a menu and the other displays a window:

- The format:

```
DBASE
```

displays the dBASE III Filing System Utilities menu.

- The format:

```
DBASE SET INDEX TO
```

displays the Indexing window.

Creating Indexes

Use the dBASE Indexing window to create a new dBASE index. Display the Indexing window by choosing the **Tools-Bonds-dBASE-Indexing** options from the Advanced Revelation Main menu, or by using the command format **DBASE SET INDEX TO** from TCL.

You may index any of the following types of fields:

- Character fields with a maximum of 100 characters
- Numeric fields
- Date fields

To specify new dBASE indexes:

1. At the *File Name* prompt enter the name of the bonded file. Do not use a file extension — the extension .DBF is assumed. After entering the file name, any indexes currently associated with the file will be displayed.
2. At the *Index Name* prompt enter the name(s) of the dBASE index(es) you wish to create. Do not use a file extension — the extension .NDX is assumed. You can specify up to nine indexes.

3. Finally, at the *Index Expression* prompt enter the name of the field to index. You cannot specify a complex expression.

Press [F9] to create the index(es).

Redefining Indexes

To redefine an index, enter a new field name at the *Index Expression* prompt of the Indexing window. You cannot specify complex expressions. The field names must be acceptable index types (see “Creating Indexes” above).

When you change indexed fields, the dBASE bond deletes the current index and rebuilds the index using the new field(s).

Disassociating Indexes

To disassociate a dBASE index from a bonded file, delete the appropriate index from the listing displayed at the *Index Name* prompt of the Indexing window.

This does not delete the .NDX file stored in the volume, but only removes the established association.

Networks

When you use the dBASE bond, all record and file locks that you execute will be honored. You will be coordinating not only with other Advanced Revelation users of the file, but with other dBASE users as well.

The dBASE III Bond Reference

The following information is important for your proper use of the dBASE bond:

Naming Conventions

The following conventions apply to all bonded dBASE files:

File Names Although the Advanced Revelation name for your bonded file can be any valid Advanced Revelation identifier, this name may not be a valid file name at the operating system level.

Consequently, when assigning a *foreign* file name for the file, the dBASE bond converts and truncates file names to conform to your operating system requirements.

- ☛ Truncation may cause a conflict with an existing file name at the operating system level.

When converting a file name, invalid characters are changed to the underscore character (_), unless the character is the

first letter of the name. In that case, the letter is changed to "A".

File Extensions Data files use the standard dBASE extension .DBF. You do not need to indicate this as part of the file name — it is appended automatically when you name or access the file.

Index files use the standard dBASE extension .NDX. As with the .DBF extension, you do not need to indicate this, since it is always assumed.

Field Names Field names have a maximum length of 10 characters. Valid names may not contain spaces and must begin with a letter. A name may consist of letters, numbers 0 through 9, and the underscore character (_).

Field names exceeding the allowable length are truncated for assignment to the dBASE file. The original Advanced Revelation field name is maintained for use inside Advanced Revelation.

Data Type Maps

These tables illustrate the data type requirements for the bond and the correspondences used to convert data types between dBASE and Advanced Revelation environments.

The following table lists the dBASE dictionary attributes.

- ☛ You can display or specify this information at the Dictionary Bonding window ([Shift-F5] from the Dictionary window).

Table M3-1

dBASE Bonding Dictionary

Type	Data	(Foreign Field Attributes)	
		Length	Decimal
C	Character	1-254	_____
N	Numeric	1-19	1-15
D	Date	8	_____
M	Memo	10*	_____
L	Logical	1	_____

*In the primary file. Occupies up to 5,000 bytes in a secondary file.

The following is the mapping of dBASE to Advanced Revelation data types. These correspondences are used by the bond when automatically constructing dictionaries:

Table M3-2
dBASE to Advanced Revelation

Type	Data	Advanced Revelation Type
C	Character	CHAR(<i>length</i>)
N	Numeric	INTEGER(<i>low, high</i>)
N	Numeric w/decimal	DECIMAL(<i>length, decimals</i>)
D	Date	DATE
L	Logical	BOOLEAN
M	Memo	VARCHAR(5000)

The following is a mapping of the standard Advanced Revelation data types to the dBASE data types. The bond applies these correspondences if you do not use the data types provided by the bond (see “Defining Files” in “Using the dBASE III Bond” above):

Table M3-3
Advanced Revelation to dBASE

Type	dBASE type	dBASE Length	dBASE Decimal
CHAR	C	< 255	_____
VARCHAR	M	10 ¹	_____
INTEGER	N	< 20 ²	_____
DECIMAL	N	< 20 ³	< 16
DOLLARS	N	< 20 ²	2
BOOLEAN	L	1	_____
DATE	D	8	_____
TIME	n/s (N) ⁴	< 20 ²	_____

Notes:

¹In the primary file. Occupies up to 5,000 bytes in a secondary file.

²Including negative sign.

³Including negative sign and decimal point.

⁴The designation “n/s” indicates that the type is not directly supported in dBASE. Data can be stored in these fields using an alternative type, but cannot be used in dBASE for the purpose suggested by the data type.













Advanced Revelation SQL Server Environmental Bond



RevelationTechnologies
August 1990



COPYRIGHT

Copyright © 1990 by Revelation Technologies, Incorporated
All Rights Reserved

No part of this publication may be reproduced by any means, be it transmitted, transcribed, photocopied, stored in a retrieval system, or translated into any language in any form, without the written permission of Revelation Technologies, Inc.

Printed in the United States of America.

TRADEMARK NOTICE

Revelation, Advanced Revelation, and Environmental Bonding are trademarks of Revelation Technologies, Inc.

Ashton-Tate, dBASE and dBASE III are trademarks of Ashton-Tate Corporation.

Microsoft, MS-DOS, SQL Server, and OS/2 are trademarks of Microsoft Corporation.

Lotus and Lotus 1-2-3 are trademarks of Lotus Development Corporation.

3COM and 3+Open are trademarks of 3COM Corporation.

Advanced NetWare and Novell NetWare Requester for OS/2 are trademarks of Novell, Inc.

TRANSACT-SQL and DB-LIBRARY are trademarks of Sybase, Inc.

SOFTWARE COPYRIGHT NOTICE

Your license agreement with Revelation Technologies, Inc. authorizes the conditions under which copies of the software can be made and the restrictions imposed on the computer system(s) on which they may be used. Any unauthorized duplication or use of any software product produced by Revelation Technologies, Inc., including Advanced Revelation, in whole or in part, in any manner, be it in print or in any other storage-and-retrieval system, is forbidden.

Advanced Revelation SQL Server Environmental Bond 1.0
ISBN 0-923387-47-1
August 1990

Chapter M 4

The SQL Server Environmental Bond

This chapter explains how to use the Advanced Revelation SQL Server Environmental Bond. The SQL Server Environmental Bond gives you access to data stored on a Microsoft SQL Server. Using the bond, you can read or write data in an SQL Server table as if it were in an Advanced Revelation file. You can also use the bond to create new tables on the SQL Server for use by Advanced Revelation.

Description of the SQL Server Bond

Components of SQL Server and the SQL Server Bond	M 4 . 1
From Advanced Revelation to SQL Server Data	M 4 . 1
Advanced Revelation SQL vs. Transact-SQL	M 4 . 2
The SQL Server Bond Environment	M 4 . 5
Characteristics of the SQL Server Bond	M 4 . 6
Advanced Revelation SQL Documentation	M 4 . 8

SQL Server Bond Requirements

SQL Server System Requirements	M 4 . 9
Workstation Requirements	M 4 . 9
Network Configuration	M 4 . 9

Setting Up the SQL Server Bond

SQL Server Bond Setup Procedure Summary	M 4 . 10
Installing the Advanced Revelation SQL Server Bond	M 4 . 10
Accessing the SQL Server	M 4 . 11
SQL Server Database Setup	M 4 . 12

Using the SQL Server Bond

Accessing SQL Server Tables	M 4 . 14
Setting Up Advanced Revelation Volumes for the SQL Server	M 4 . 15
Using the SQL Server Access Window	M 4 . 17
Using Existing SQL Server Files	M 4 . 19
Defining New SQL Server Files	M 4 . 23
Updating SQL Server Files	M 4 . 26
Copying Files and Records	M 4 . 27
Modifying the File Structure	M 4 . 28

(continued)

Using Existing SQL Server Views	M 4 . 30
Creating SQL Server Views	M 4 . 32
Planning Security	M 4 . 32
Indexing	M 4 . 37

The SQL Server Bond Reference

Advanced Revelation SQL Commands and Keywords	M 4 . 43
R/BASIC Programs	M 4 . 45
Naming Conventions	M 4 . 45
Record Structure	M 4 . 49
Truncation Handling	M 4 . 55
Advanced Revelation Validation Patterns and SQL Server Rules	M 4 . 56
Default Column Values	M 4 . 57
SQL Server Triggers	M 4 . 59
SQL Server Stored Procedures	M 4 . 60
Null Handling	M 4 . 61
Locking	M 4 . 63
Transaction Processing	M 4 . 64
Error Handling	M 4 . 65
Using the MSSQL_LOG File	M 4 . 67
Freeing Memory	M 4 . 68

MSSQL_C Routine

Function 1 - Function 24	M 4 . 68
About the ERRFLAG Parameter	M 4 . 83
About the ERRSTR Parameter	M 4 . 87
Examples of Function Usage	M 4 . 87
DB-LIBRARY Functions and Macros Used	M 4 . 89

NETSERVERENUM Routine

NETSERVERENUM Routine	M 4 . 89
---------------------------------	----------

Description of the SQL Server Bond

Advanced Revelation's SQL Server bond gives you a direct, interactive, and transparent connection to data stored on a Microsoft SQL Server. With the SQL Server bond, you can treat SQL Server tables just as you would any other files in your Advanced Revelation system. This includes the ability to use Advanced Revelation's full range of application development tools. The structure of SQL Server tables is not affected by the SQL Server bond. You can access SQL Server tables from Advanced Revelation and from other environments simultaneously.

Components of SQL Server and the SQL Server Bond

Microsoft SQL Server/SQL Server Microsoft SQL Server, referred to throughout this documentation as the SQL Server, is a collection of programs written by Microsoft and Sybase. As a group, these programs give users direct access to data stored on Microsoft SQL Servers, which run under OS/2, or Sybase servers, which run under several mini/micro computer platforms.

SQL Server and other programs of its type are known as database servers. A database server ("back-end server") acts as middle man between database management systems like Advanced Revelation ("front-ends" or "clients") and data. As a go-between, SQL Server enables you to augment the advantages of your front-end applications with the benefits of centralized processing: faster data access, accessibility of data to multiple front-ends, and greater data integrity.

SQL Server Bond If the SQL Server is the middle man between Advanced Revelation and data, then the SQL Server Bond is the middle man between Advanced Revelation and SQL Server. The SQL Server Bond enables you to access SQL Server tables and views using Advanced Revelation tools without ever being aware of SQL Server.

From Advanced Revelation to SQL Server Data

Advanced Revelation, the SQL Server bond, and SQL Server interact to simplify data entry and retrieval, enforce data integrity, and maintain system security. A general understanding of how Advanced Revelation accesses SQL Server data by way of the SQL Server Bond and SQL Server will help you understand their relationship to one another.

No matter how you access SQL Server data, the route from Advanced Revelation to that data is the same:

1. Within Advanced Revelation, execute a process that accesses SQL Server data (an I/O process). This includes any of the standard Advanced Revelation operations:
 - Edit a row (record) in an Advanced Revelation data entry window.
 - Run an R/LIST report.
 - Execute an SQL statement from the Advanced Revelation interactive SQL window.
 - Run an R/BASIC program that contains an embedded SQL statement, R/BASIC READ or WRITE statement, or any other file access statement.
2. The Advanced Revelation SQL Server bond translates the operation into the language understood by the SQL Server — Transact-SQL.
3. If step 1 was a request for data, SQL Server retrieves the data and sends it back to Advanced Revelation. If step 1 was any other process (for example, an SQL INSERT, UPDATE, or DELETE statement), SQL Server sends back either a “process complete” message or an error message.

The diagram on the next page maps the route from Advanced Revelation processes to data in the Advanced Revelation, SQL Server, and other environments.

Advanced Revelation SQL vs. Transact-SQL

Advanced Revelation SQL and Transact-SQL are both implementations of ANSI (American National Standards Institute) SQL. However, the ANSI standard is just a starting point for both, and each includes a number of enhancements to standard SQL. These enhancements mean some minor differences in command names and formats between the two versions. In addition, Transact-SQL includes an array of system administrator commands and other enhancements that have no direct analog in Advanced Revelation and that affect only the SQL Server environment.

You can execute Transact-SQL commands without leaving Advanced Revelation by using the Advanced Revelation Server Access Window (SAW).

- ☛ Transact-SQL commands executed from the SAW window operate only in the SQL Server environment and, therefore, affect only SQL Server data. If you attempt to run a Transact-SQL command that accesses a file or table in the Advanced Revelation environment, SQL Server will return a message indicating that the table is not available.

Table 1 identifies the parallels between Advanced Revelation SQL and Transact-SQL. Commands that appear only in the Transact-SQL column must be executed from the SQL Server Access Window.

- ☛ For information on the SAW window, see “Using the SQL Server Access Window” later in this chapter.

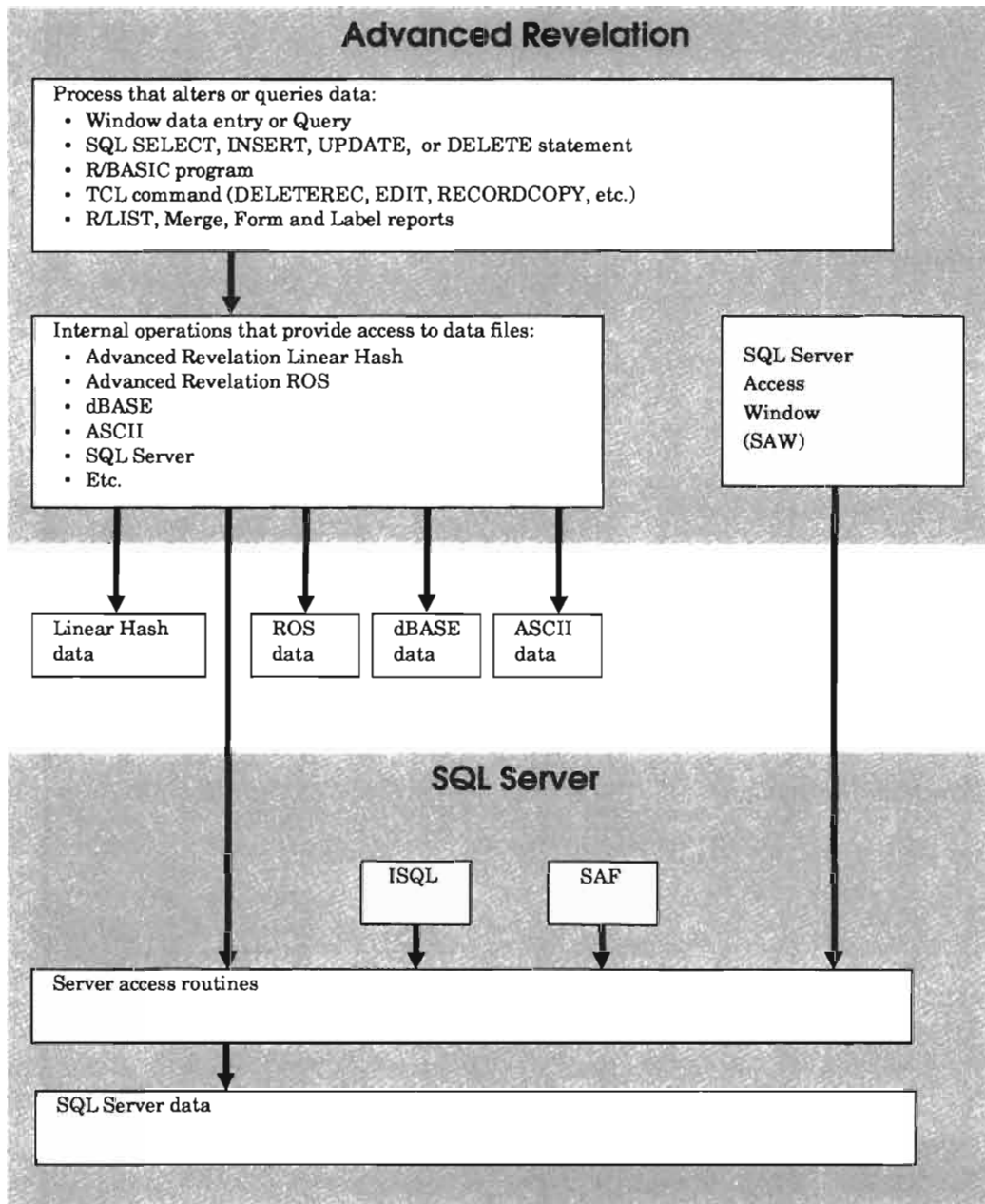


Table M4-1

Advanced Revelation SQL vs. Transact-SQL

	Advanced Revelation SQL	SQL Server Transact-SQL
Data definition	ALTER TABLE ¹ CREATE TABLE DROP TABLE CREATE VIEW ² DROP VIEW ²	ALTER TABLE ¹ CREATE TABLE DROP TABLE CREATE VIEW ² DROP VIEW ² ALTER DATABASE CREATE DATABASE CREATE DEFAULT CREATE INDEX CREATE RULE CREATE TRIGGER DROP DATABASE DROP DEFAULT DROP INDEX DROP RULE DROP TRIGGER User-defined Data Types
Data manipulation	DELETE INSERT UPDATE	DELETE INSERT UPDATE CREATE PROCEDURE DROP PROCEDURE TRUNCATE TABLE
Security		GRANT REVOKE SETUSER USE
Queries	SELECT	SELECT (continued)

Table M4-1

Advanced Revelation vs. Transact-SQL *(continued)*

	Advanced Revelation SQL	SQL Server Transact-SQL
Transaction processing	SET TRANSACTION BEGIN TRANSACTION COMMIT {WORK} ROLLBACK {WORK}	BEGIN TRANSACTION COMMIT TRANSACTION ROLLBACK TRANSACTION SAVE TRANSACTION
Advanced Programming language	R/BASIC	Control-of-flow language
System Administration		CHECKPOINT DUMP TRANSACTION LOAD TRANSACTION (backup, system stored procedures, etc.)

¹The Advanced Revelation SQL command ALTER TABLE cannot alter the structure of a table on the SQL Server. See "Modifying the File Structure" for more information.

²The Advanced Revelation SQL commands CREATE VIEW and DROP VIEW operate exclusively on Advanced Revelation views. You cannot use Advanced Revelation SQL to create or drop a view on the SQL Server.

The SQL Server Bond Environment

The SQL Server Bond provides an additional menu and window as extensions to the standard Advanced Revelation environment.

The SQL Server Bond Menu To display the SQL Server Bond menu, choose the **Tools-Bonds-SQL Server** options from the Advanced Revelation Main menu.

The SQL Server Access Window (SAW) To display the SQL Server Access window (SAW), choose the single option **Server Access** at the SQL Server Bond menu or enter "SAW" at TCL.

Characteristics of the SQL Server Bond

The client-server relationship of Advanced Revelation and the SQL Server requires that you keep a few issues in mind when using the SQL Server Bond. Details about characteristic listed below appear in the “Using the SQL Server Bond” section and in the “SQL Server Bond Reference” section.

Copying Files In the current release of the SQL Server Bond, you cannot use the TCL FILECOPY command to copy a table. Instead, use the SQL Server’s bulk copy utility.

Creating UNIQUE Columns When using the Advanced Revelation SQL CREATE TABLE command or using the Create Table window at the Advanced Revelation SQL window, do not use the UNIQUE key word. Also, at the Dictionary Bonding Information window do not set *Unique* to yes. UNIQUE is not supported in this version of the SQL Server Bond. If you want to specify that a column is unique, you must use Transact-SQL to make this specification directly on the SQL Server.

Data Types The SQL Server TIMESTAMP data type is ignored by Advanced Revelation. You cannot read or display the timestamp for SQL Server rows.

To use the SQL Server IMAGE data type or the Advanced Revelation VARBINARY data type successfully, you must comply with these restrictions: there can be only one IMAGE column or one VARBINARY column per table; the IMAGE or VARBINARY column must be the last column in the record.

Indexing You can use SQL Server indexes with your bonded SQL Server tables. Adding Advanced Revelation indexes to your bonded SQL Server tables is *not* recommended. Both within and outside of Advanced Revelation, there are table update processes that may bypass the Advanced Revelation indexing system, causing an Advanced Revelation index to be out of date.

The SQL Server Bond does not support Quickdex/Rightdex indexes. This index type would violate domain constraints of an SQL Server table because it adds a %RECORDS% field to the table, altering the table structure.

The Advanced Revelation subroutines BTREE.EXTRACT and COLLECT.IXVALS are not supported by the SQL Server indexes. These are strictly Advanced Revelation indexing (SI.MFS) access routines.

Locking The SQL Server manages locking for *all* users accessing data on the SQL Server.

R/BASIC LOCK calls will use the network locking that linear hash files use.

In transactions involving SQL Server tables, locking is pre-set to consistency level 3 (exclusive locks held for the duration of the transaction during modifications and shared locks held during queries). The SET TRANSACTION CONSISTENCY LEVEL statement will not affect the locking of SQL Server tables.

Modifying the File Structure

When the structure of an SQL Server table changes, you must delete and recreate the existing Advanced Revelation dictionary for the table.

Currently, you *cannot* use any Advanced Revelation tool to alter the structure of a table on the SQL Server. However, you can use the Transact-SQL ALTER TABLE command at the SQL Server Access Window.

MSSQL_LOG File

When an Advanced Revelation process causes an error on the SQL Server, information about the error is logged in a file called MSSQL_LOG. If server tracing is turned on, the MSSQL_LOG will also contain records of Transact-SQL commands that Advanced Revelation processes have sent to the SQL Server. Over time the MSSQL_LOG file may grow to a considerable size. Periodically you should clear the MSSQL_LOG file to reduce its size and free up space on your hard disk.

Multipart Keys

SQL Server limits keys to 8 parts.

The combined length of all columns in a multipart key may not exceed 256 characters. The bond always attempts to create a composite clustered index on the SQL Server for the columns that are designated as record key parts. SQL Server limits clustered and composite indexes to a maximum of 256 characters.

Searching for Strings Using a LIKE Clause

When searching for strings using the LIKE clause, do not use the ESCAPE clause to replace the default “\” with another character. Use of the ESCAPE clause with the SQL Server Bond has not been implemented.

SQL Cursors

The use of Advanced Revelation SQL cursors with the SQL Server Bond has not been implemented.

Transaction Processing

The SQL Server Bond makes use of the 2 Phase Commit utility on the SQL Server.

If a system failure occurs during the commit phase, you must coordinate recovery activities on the SQL Server and in Advanced Revelation.

Truncation Handling

Advanced Revelation truncation handling is not implemented in the SQL Server Bond. The SQL Server will automatically truncate data that is in excess of the length defined for a column. For example, if you have defined a column as data type CHAR(3) and attempt to update that column with the value 1234, the SQL Server will automatically truncate the value when it is saved in the database. If you select the updated row from the database, the value returned for that column will be 123.

In addition, if you access a row on the SQL Server that is longer than 64K, Advanced Revelation will automatically truncate the row to 64K, and you *will* lose data.

The specifications at the *Acceptable Data Loss* prompts in the Dictionary Bonding Information window are ignored by the SQL Server Bond. Consequently, if data is too long or too precise for the field, the write operation may succeed (it will not be rejected), but you may experience partial or total data loss. Use of SQL Server domain validation and/or Advanced Revelation data validation patterns can eliminate this problem.

Updating Tables and Views with Advanced Revelation Tools

The ability to update tables on the SQL Server is controlled by permissions granted on the server. In order to use Advanced Revelation tools to update a table or a view on the server, you must have full access/update permission on the table(s).

Views

If you want to use Advanced Revelation tools to modify data in views, you must have full SQL Server permission to update, insert, or delete data in the base table(s). If this is not the case, you must use Transact-SQL commands to make your updates.

Advanced Revelation SQL Documentation

See *SQL* for full documentation of Advanced Revelation SQL.

SQL Server Bond Requirements

In order to use the Advanced Revelation SQL Server Bond successfully, your system must comply with the hardware and software requirements and network configuration described below.

SQL Server System Requirements

Your system must meet the minimum SQL Server requirements specified in the Microsoft SQL Server installation documentation.

- ☛ We recommend that the computer running the SQL Server be dedicated for exclusive use as the SQL Server. In other words, it should not also serve as a workstation.
- ☛ Note that a typical SQL Server environment, which includes OS/2, LAN Manager network, and SQL Server software, occupies approximately 36 MB on the server hard disk.

Workstation Requirements

Your SQL Server workstations must run MS-DOS. Advanced Revelation for OS/2 version 2.01 does not support the current release of the SQL Server Bond. MS-DOS workstations must meet the minimum requirements specified in the Microsoft SQL Server installation documentation. In addition, to use the Advanced Revelation SQL Server Bond your MS-DOS workstations must meet the requirements listed below.

- Must have expanded memory capabilities (minimum of 1 MB expanded memory).
- Must be running MS-DOS version 3.1 or higher.
- Must be running the MS-DOS version of Advanced Revelation 2.02 or higher.
- Must be running software that allows you to communicate with the server over the network. For example, MS-DOS workstations on LAN Manager-compatible networks support named pipes communication protocol and must run the TSR program DBNMPPIPE.EXE (Resident Named Pipe Network Interface for DB-Library).

Network Configuration

To run the SQL Server, your network software and network configuration must conform to the requirements specified in the Microsoft SQL Server installation documentation. Advanced Revelation supports all the networks that can run with the SQL Server.

- ☛ Revelation Technologies has tested and successfully run the SQL Server Bond on these networks:

LAN Manager
3COM 3+ Open
Advanced NetWare with the SQL Server using Novell NetWare
Requester for OS/2

Setting Up the SQL Server Bond

This section provides information about setting up the Advanced Revelation SQL Server Bond.

SQL Server Bond Setup Procedure Summary

1. Install the Advanced Revelation SQL Server Bond on your network copy of Advanced Revelation version 2.02 or higher. See “Installing the Advanced Revelation SQL Server Bond” below for details.
2. Boot the SQL Server and log onto Advanced Revelation. See “Accessing the SQL Server” below for details.
3. Complete the SQL Server Database Setup. See “SQL Server Database Setup” for details.
4. Set up at least one Advanced Revelation volume for the SQL Server. See “Setting Up Advanced Revelation Volumes for the SQL Server” in “Using the SQL Server Bond” for details.

Installing the Advanced Revelation SQL Server Bond

- ☛ You must install the SQL Server Bond on Advanced Revelation for MS-DOS version 2.02 or higher. If you are currently running an earlier version of Advanced Revelation, you must upgrade your copy of Advanced Revelation to version 2.02 or higher *before* installing the SQL Server Bond.

To install the SQL Server Bond:

1. Use the installation procedure for optional modules. See the “Updating and Enhancing Your Advanced Revelation System” chapter in *Installation* for details on this process.
2. Run the Upgrade Account process to make the SQL Server bond available to existing user accounts in your Advanced Revelation system. See the “Updating and Enhancing Your Advanced Revelation System” chapter in *Installation* for details on this process. (You do not need to run the Upgrade Account process if you are installing Advanced Revelation for the first time or if you have not created user accounts in Advanced Revelation.)

- ✎ After installing the SQL Server Bond, log onto Advanced Revelation as described in “Accessing the SQL Server” below. You *must* then run the procedures described in “SQL Server Database Setup”. Setup must be completed before you can use the Advanced Revelation SQL Server Bond successfully.

Accessing the SQL Server

Once your Advanced Revelation SQL Server Bond is installed, use a procedure like the one listed below to access the SQL Server:

1. To boot the SQL server:
 - Boot the server computer. OS/2 starts up.
 - Boot your network.
 - Start the SQL Server using one of the ways described in your SQL Server system administrator documentation. The method you use depends on the network software you are running.
2. Boot the MS-DOS workstation as you normally do. Your bootup process must log you onto the network and load software that allows you to communicate with the SQL Server over the network. For example, if you are running on a LAN Manager network, bootup should load the TSR program DBNMPPIPE.EXE, which allows you to use the network to access the SQL Server. The TSR program DBNMPPIPE.EXE (Resident Named Pipe Network Interface for DB-LIBRARY) can be loaded as part of your AUTOEXEC.BAT file processing.
 - ✎ The DBNMPPIPE.EXE and ENDDBLIB.EXE programs are provided on the SQL Server Bond Optional Module disk. If needed, you can copy them to your hard disk. For more information about ENDDBLIB.EXE see “Freeing Memory” in the “Reference” section.
3. Log onto Advanced Revelation as you would for any network application. If yours is one of several MS-DOS workstations sharing Advanced Revelation files on the network, you should all use the same AREV.EXE to log on. We recommend that your logon command include the X logon option to invoke expanded memory support in Expanded mode. For example:

```
AREV SYSPROG /X
```

The command shown above logs the user into the SYSPROG account in Advanced Revelation and specifies that expanded memory is used in Expanded mode. For more information about logon syntax and expanded memory, see the appropriate chapters in *Installation*.

SQL Server Database Setup

To perform Advanced Revelation file access operations in a database on the SQL Server, you *must* use the stored procedure **sp_dboption** to turn on the SELECT INTO/BULKCOPY database option for that database. The system administrator should use the **sp_dboption** stored procedure on:

- All existing SQL Server databases that will be accessed from Advanced Revelation.
- The MODEL database, to ensure that future databases have the SELECT INTO/BULKCOPY option on.

Details on running the **sp_dboption** stored procedure in Advanced Revelation appear below.

- ☛ See your SQL Server Language documentation for additional information about using **sp_dboption**. Note that turning on the SELECT INTO/BULKCOPY option may restrict the use of some SQL Server commands.

You can execute **sp_dboption** and other stored procedures at the SQL Server Access window (SAW) in Advanced Revelation. To reach the SAW window:

1. Select **Tools-Bonds-SQL Server-Server Access** from the Advanced Revelation Main menu.
2. The SQL Server Login window displays, prompting you for information that is used to log on to the SQL Server.

To perform the necessary SQL Server database setup procedures, you must log on as the system administrator. Enter:

- The name of the SQL Server that you want to use.
- The login name "SA".
- The appropriate password.

Press [F9] to log on to the SQL Server and display the SAW window.

Turning on SELECT INTO/BULKCOPY in MODEL

By default, the SELECT INTO/BULKCOPY option is off in newly created databases. To change the default setting, use the stored procedure **sp_dboption** to set this option *on* in the MODEL database. This will ensure that future databases have the SELECT INTO/BULKCOPY option on.

Issue these individual commands at the SAW window:

```
USE MASTER
SP_DBOPTION MODEL, "SELECT INTO/BULKCOPY", TRUE
USE MODEL
CHECKPOINT
```


- Each time you enter one of the commands listed above, press [F9] to execute it before entering the next command.

For more information about using the SAW window, see "Using the SQL Server Access Window" in this chapter.

Turning on SELECT INTO/BULKCOPY in Existing SQL Server Databases

You must turn on the SELECT INTO/BULKCOPY database option for the existing SQL Server databases that you want to access from within Advanced Revelation. For each individual SQL Server database that you want to access from Advanced Revelation, issue the following individual commands at the SAW window:

```
USE MASTER
SP DBOPTION database_name,
"SELECT INTO/BULKCOPY", TRUE
USE database_name
CHECKPOINT
```

- Each time you enter one of the commands listed above, press [F9] to execute it before entering the next command.

For more information about using the SAW window, see "Using the SQL Server Access Window" in this chapter.

You only need to issue these commands once for each database. Once you have turned on the SELECT INTO/BULKCOPY option for a database, it remains on until you reset the option. Use these commands only for databases that were created prior to turning on the SELECT INTO/BULKCOPY option in MODEL.

Using the SQL Server Bond

Working with a bonded SQL Server table is very similar to using any other Advanced Revelation file. You can use most of the same tools and perform most standard operations on SQL Server tables.

- ☞ The “Introduction to Environmental Bonding” chapter in *Environmental Bonding* provides background information about standard Environmental Bond behavior.

In addition to allowing you to use Advanced Revelation tools to create and maintain databases on the server, the SQL Server Bond provides a specialized tool, the Advanced Revelation SQL Server Access window (SAW), that allows you to accomplish tasks that are unique to the SQL Server. These tasks are accomplished using the SQL Server’s native language, Transact-SQL. You can use the SAW window to enter, edit and execute Transact-SQL commands from within Advanced Revelation. The Transact-SQL commands that you enter at the SAW window apply *only* to databases on the SQL Server.

- ☞ For more information about the SAW window, see “Using the SQL Server Access Window” in this chapter.

Any requirements or distinctive operations involved in using the SQL Server bond are explained below. In all other circumstances, if your system is configured as described in the “SQL Server Bond Requirements” section and if the bond is properly installed, you should assume that a bonded SQL Server table requires no special treatment.

Accessing SQL Server Tables

You must establish a volume for each individual database that you want to access on the SQL Server. See “Setting Up Advanced Revelation Volumes for the SQL Server” below for information about creating an SQL Server volume definition.

An SQL Server volume definition tells Advanced Revelation:

- The name of the server and database where the SQL Server files are stored and the file type.
- The login ID and password required by SQL Server security in order to access the database.

After you have assigned a volume for your SQL Server files, you must attach the volume in order to make the files available to Advanced Revelation processes. Choose the **Tools-Files-Attach** options from the Advanced Revelation Main menu, or use the ATTACH command from TCL.

- ☞ For more information about volumes see the chapters “Advanced Revelation Volumes” in *Filing Systems* and “Introduction to Bonding” in *Environmental Bonding*.

You can create multiple volumes if you have SQL Server tables in more than one database, if you have access to multiple SQL Servers, or if your security requirements warrant separate logon procedures for different users.

- ☞ If the security of the data in an SQL Server database is of great concern, you may need to set up multiple volume definitions to accommodate multiuser access to a database. For more information about using volumes to comply with SQL Server security, see “Planning Security” later in this section.

Setting Up Advanced Revelation Volumes for the SQL Server

Each SQL Server volume that you set up gives you access to a specific database on the SQL Server. Use the Set Volume window to create a volume. Follow these steps:

1. Use the menu options **Tools-Files-Set Volume** to display the Set Volume window.
2. At the *Volume Name* prompt, enter a name for the volume you are establishing. Do not use spaces in the name.
3. At the *File Type* prompt, enter MSSQL_BFS to identify the bond you are using. You can press [F2] to see a list of available file types.
4. At the *Data Location* prompt, press [F2] to call the MS-SQL Volume Attributes window.

The MS-SQL Volume Attributes window collects information that the bond requires in order to pass through security on the SQL Server. In addition, it enables you to turn on a server trace feature so that you can record the Transact-SQL queries that are generated by Advanced Revelation processes. Information entered at the MS-SQL Volume Attributes window tells the SQL Server Bond:

- Which SQL Server to log onto.
- Which database to use.
- SQL Server login ID.
- SQL Server password (optional).
- Whether or not to trace all Transact-SQL queries generated by Advanced Revelation processes.

The volume attributes information becomes part of the volume definition. Any time that you attach the volume, the SQL Server bond will use this information to log you onto the specified database on the server, making the appropriate tables in the database available to you.

- ☞ If your SQL Server is defined as case sensitive, be careful to fill out the prompts for database name, login ID and password precisely as they are defined on the SQL Server.
5. At the *Server Name* prompt, enter the name of the SQL Server you want to use. Each SQL Server is assigned a unique name as part of that server's installation process. You can press [F2] to select from a popup list of available SQL Servers.

☞ When you press [F2] at the *Server Name* prompt, your network broadcasts a call to see what servers are currently available. Servers that respond are then listed at the popup. However, any server that is actively processing data when the call is made will not be free to respond. Therefore, your popup list of servers may not always be a complete one. You may enter any valid server name at the *Server Name* prompt, even one that is not listed at the popup.

6. At the *Database Name* prompt, enter the name of the database on the SQL Server that this volume will access.

☞ An SQL Server volume enables you to access, add, and update tables in *existing* databases. You cannot use the volume definition to create a database on the server. Creating databases is a function performed by the SQL Server system administrator, using Transact-SQL commands. Within Advanced Revelation, Transact-SQL commands can be entered at the SQL Server Access window (SAW).

7. The volume definition must provide an established SQL Server login ID that the bond can use to log you into the database.

At the SQL Server *Login Name*, there are three ways to provide the correct SQL Server login ID. You can:

- Enter the appropriate login ID.
- Direct Advanced Revelation to prompt the user for a login ID.
- Direct Advanced Revelation to use the name that the user entered when logging onto Advanced Revelation for the current session.

Press [F2] at the *Login Name* prompt and use the Login Name Options popup to specify how the SQL Server login ID will be provided.

☞ For more detailed information about using logins to comply with SQL Server security, see the topic “Logins and Passwords” under “Planning Security” later in this section.

8. If there is an SQL Server password assigned to the specified login ID, the volume definition must provide the password so that the bond can log you into the database. A password is required only if the SQL system administrator assigned one for use with your log-in ID.

There are three ways to provide the correct password. At the *Password* prompt you can:

- Enter the appropriate password.
- Direct Advanced Revelation to prompt the user for a password.
- Direct Advanced Revelation to use the password that the user entered when logging onto Advanced Revelation for the current session.

Press [F2] at the *Password* prompt and use the Password Options popup to specify how the password will be provided.

For more detailed information about using passwords to comply with SQL Server security, see the topic “Logins and Passwords” under “Planning Security” later in this section.

9. If you want to review the Transact-SQL queries that are generated by Advanced Revelation processes, enter Y at the *Server Tracing* prompt. This turns on the Advanced Revelation server trace function for this SQL database. Server trace records all Transact-SQL queries generated by Advanced Revelation processes working in this volume.
 - ☛ For more information about the server trace function, see “Using the MSSQL_LOG File” in the “Reference” section.
10. Press [F9] to save your volume attribute specifications and return to the Set Volume window.
11. At the *Control Files Location* prompt, enter the drive or subdirectory name where you want to store supporting Advanced Revelation files such as dictionaries. You can choose any location for your control files except the SQL Server itself. If a control file location is not specified, the default control files location is *path*\MSSQL.BND, where *path* is the path from which you logged onto Advanced Revelation. For example, if you logged onto Advanced Revelation from E:\AREV, the default control files location for the SQL Server Bond is the subdirectory E:\AREV\MSSQL.BND.
 - ☛ Control files are described in detail under the topic “Control Files” in the “Introduction to Bonding” chapter in *Environmental Bonding*.
12. Press [F9] to save your volume specifications.

Using the SQL Server Access Window

You can issue individual Transact-SQL commands directly to the SQL Server from within Advanced Revelation by using the SQL Server Access window (SAW).

- ☛ The Transact-SQL commands that you enter at the SAW window apply *only* to databases on the SQL Server. You cannot enter Advanced Revelation TCL, SQL, or R/BASIC commands or work with Advanced Revelation files at this window.

To reach the SAW window:

1. Select **Tools-Bonds-SQL Server-Server Access** from the Advanced Revelation Main menu.
2. The SQL Server Login window displays. The SQL Server Bond uses the information that you enter at this window to log you into the appropriate SQL Server.

At the *Server Name* prompt, enter the name of the SQL Server you want to use. Each SQL Server is assigned a unique name as part of that server's installation process. You can press [F2] to select from a popup list of available SQL Servers.

- ☛ When you press [F2] at the *Server Name* prompt, your network broadcasts a call to see what servers are currently available. Servers that respond are then listed at the popup. However, any server that is actively processing data when the call is made will not be free to respond. Therefore, your popup list of servers may not always be a complete one. You may enter any valid server name at the *Server Name* prompt, even one that is not listed at the popup.

At the SQL Server *Login Name*, enter your login ID on the SQL Server.

If there is an SQL Server password assigned to the specified login ID, enter it at the *Password* prompt. A password is required only if the SQL Server system administrator assigned one for use with your login ID.

After entering the required information, press [F9], or press [Enter] at the *Password* prompt, to log into the SQL Server.

- ☛ If your SQL Server is case sensitive, be careful to fill out the prompts for login name and password precisely as they are defined on the server.

3. The SAW window displays. To work with a database other than the default database associated with your SQL Server login ID, enter the Transact-SQL command "USE *database_name*" and press [F9].

Active Keys The SAW window operates just like any other window in Advanced Revelation. It uses Advanced Revelation's standard edit and function keys. Enter commands at the SAW window just as you would at TCL. Here is a summary of important active keys:

- [F9] Save. Each time you enter a command, press [F9] to execute it.
- [F2] Command stack. Press [F2] to display a popup that contains the last 20 Transact-SQL commands that have been entered during the current session at the SAW window. A command can be selected from the popup and returned at the SAW window. The command stack is cleared each time you exit the SAW window.

- [Shift-F1] Store the current command. Transact-SQL commands entered at the SAW window can be stored as records in a file called SAW_SCRIPTS. If you would like to save the current command so that it can be run again, press [Shift-F1]. You will be prompted to enter a name for the command. The name that you enter serves as the key to the command record in the SAW_SCRIPTS file.
- [Shift-F2] Retrieve a stored command. Press [Shift-F2] to retrieve a stored command. A popup of existing records in the SAW_SCRIPTS file displays. When you choose a record from the popup, the Transact-SQL command is returned at the SAW window.
- [Shift-F3] Direct output to the printer. Press [Shift-F3] to send the results of the next operation to the printer. For example, you can send the results of a Transact-SQL SELECT statement to the printer by pressing [Shift-F3] and then entering and executing the SELECT operation. At the completion of the operation, output reverts to the screen. You would press [Shift-F3] again to send the results of the next operation to the printer.

Operating Multiple SAW Windows

You can open more than one SAW window by going from one TCL level to another. Occasionally, when you operate multiple SAW windows, you may experience a lock wait situation. A lock wait will occur if you begin a transaction in one SAW window, and then open a second SAW window and attempt to perform an operation on records that are locked by the uncommitted transaction in the original SAW window. The process in the second SAW window will wait for a lock *only* as many seconds as are specified at the *Lock Timeout* prompt at the Concurrency window. At the end of this time, the command at the second SAW window is aborted.

Using Existing SQL Server Files

Complete the steps listed below to use an existing SQL Server file in Advanced Revelation.

- 1. You only need to perform steps 1 and 2 once for any SQL Server database that you want to access. You must always perform step 3 (attaching files) during an Advanced Revelation session in order to use tables in an SQL Server database.

1. To successfully access the tables in an SQL Server database from within Advanced Revelation, the SQL Server SELECT INTO/BULKCOPY database option for that database must be on. *If you have not already done so, you should turn on this option. For details about SELECT INTO/BULKCOPY, see “SQL Server Database Setup” in this chapter.*
2. Create a volume for the database on the SQL Server that contains the table(s) you want to use. See “Setting Up Advanced Revelation Volumes for the SQL Server” above for details on this process.
3. Attach the SQL Server volume in order to make the tables in the volume available to Advanced Revelation processes. Choose the **Tools-Files-Attach** options from the Advanced Revelation Main menu, or use the ATTACH command at TCL.

☞ If you want certain files to be available every time that you log into Advanced Revelation, you should include the ATTACH command in your logon script. See the chapter “Users & Accounts: Planning Access and Security” in *Developing an Application* for more information.

Building the Advanced Revelation Dictionary

The first time you use an Advanced Revelation tool to access an existing SQL Server table, an Advanced Revelation dictionary is dynamically created for the file. The existing columns in the SQL table are mapped to corresponding dictionary definitions in Advanced Revelation.

☞ For more information, see “Data types: Attaching SQL Server Tables” in the “Reference” section in this chapter.

The dictionary created for a file is assigned the standard Advanced Revelation identifier of:

`DICT.filename`

where *filename* is the name of the existing SQL Server table. To display the dictionary, use the TCL command LISTDICT.

About Record Keys in Existing Files

The bond uses SQL Server indexing information to determine which field is the record key.

☞ For an in-depth look at SQL Server indexing, see “Indexing” later in this section.

To designate key field(s), the bond determines what is the lowest numbered unique index in SYSINDEXES. The column(s) associated with that unique index is defined as the record key in the Advanced Revelation dictionary.

If you do not have a unique index assigned to any of the columns in a table, the SQL Server Bond cannot recognize a key field. Consequently, the table will operate as an SQL Server view within Advanced Revelation. Because it operates as a view, you will not be able to access or update this table by any Advanced Revelation process that requires a key (for example, data entry window operations). For more information about working with views, see “Using SQL Server Views” later in this section.

- ☛ If an existing table is being treated as a view in Advanced Revelation, you may want to assign an SQL Server unique index to the key column and then to rebuild the Advanced Revelation dictionary. Once the dictionary is rebuilt, you will have full use of the table in Advanced Revelation. It will no longer be treated as a view. See “Assigning a Record Key for an Existing File” below for more information.

Checking the Record Key in an Existing File

Use the LISTDICT command to determine which column(s) is defined as the record key. At TCL enter:

```
LISTDICT filename
```

This produces a dictionary listing of all of the fields in the current file. To determine the key to the file, look at the “Pos” column in the listing:

- The key to the record is always assigned field position 0 (zero).
- For multipart keys, each field that is a part of the key will be position 0*n, where *n* is the key part number. The bond creates a multipart key in the dictionary if it finds that there is a composite clustered index on the SQL Server table.
- If none of the listed fields has position 0, the bond could not identify a key field. This indicates one of two things:
 - There is not a unique index assigned to any column in this table. The table will operate as a view.
 - This database object is a view.

Assigning a Record Key for an Existing File

To be able to use all Advanced Revelation tools on an SQL Server table, there must be *at least one SQL Server unique index* assigned to a column in the table. This ensures that the SQL Server Bond can designate a record key field in the Advanced Revelation dictionary.

If the bond will access an existing SQL Server table, you should create a unique clustered index for the column(s) that you want to use as the primary key(s) (subject to the SQL Server's 256 character length limitation on entries in clustered indexes).

Ideally, the unique index should be created for the table *before* you access the table in Advanced Revelation. If the bond does not find a unique index, the dictionary for the SQL Server table will not contain a key field and the table will operate as a view.

If you discover that the dictionary for an SQL Server table does not contain a key field, and if you do not want the table to be treated as a view in Advanced Revelation, complete these steps to assign a record key:

1. Use Transact-SQL to assign an SQL Server unique index to the key column(s) for the table. Usually this will be a unique clustered index.
2. At TCL enter the command:

```
DELETEDFILE DICT.tablename
```

to delete the current Advanced Revelation dictionary for the table.

3. At TCL enter the command:

```
ATTACH volumename
```

to reattach the volume so that Advanced Revelation can recognize the changes that have been made.

4. At TCL enter the command:

```
LISTDICT tablename
```

Because no dictionary exists, one will be generated automatically. The new dictionary will reflect the presence of a unique index. The column(s) that you indexed will be the key(s) to the record.

Modifying the Dictionary

Advanced Revelation, when creating a dictionary for an existing SQL Server table, sets several values for each column. These values include *Dict type*, which is set to “F”, and *Master definition*, which is set to “Yes”. The values of *Dict type* and *Master definition* cannot be changed. In addition, you can not change the existing *Data type*, *Length*, and *Value required* (Null/Not Null) specifications for any column.

☛ For more information, see “Modifying the File Structure” in this chapter.

Adding Dictionary Records

If you need type G (Group) or type S (Symbolic) fields or field synonyms, just define additional fields in the Advanced Revelation dictionary. There is no restriction on adding these types of fields.

The bond *will not* permit you to add new “master” fields (fields in which the *Master definition* specification is set to yes). You can add a type F field to the dictionary for an existing file *only* if the field that you are adding is a synonym.

Deleting Dictionary Definitions

Deleting an existing column is not permitted by the SQL Server. The bond *will not* permit you to delete existing “master” fields (fields in which the *Master definition* specification is set to yes).

You may delete type G (Group) or type S (Symbolic) fields or field synonyms at any time.

Defining New SQL Server Files

You can use any of the following Advanced Revelation file creation tools to create a table on the SQL Server:

- Define File window. For more information on the Define File window, see the chapter “Managing Files and Records” in the *Filing Systems* section.
- SQL Assistant’s Create Table window. The SQL Assistant is documented in “The SQL Option” chapter of *SQL*.
- Advanced Revelation SQL CREATE TABLE command. The CREATE TABLE command is documented in the *SQL* section.
- Paint. Paint is described in detail in the section *Developing an Application*.
- TCL DEFINEFILE command. The DEFINEFILE command is documented in the “TCL Command Reference” chapter of *TCL*.

Choose the tool that you prefer. For example, if you are unsure of all the options you have when defining a file, use the Define File window. If you are familiar with SQL, you might choose the SQL CREATE TABLE command instead.

Both the Define File window and Paint offer development flexibility since they allow you to define the dictionary for a file over multiple sessions. The table is not actually created on the SQL Server until you indicate that you want to commit to the current file structure.

The prompts at the SQL Assistant CREATE TABLE window reflect the structure of an SQL CREATE TABLE command. So when you define fields with this tool it is particularly easy to set a default value and/or specify that the column accepts nulls.

Whatever method you choose to create a file, the end result is the same. All of the Advanced Revelation file creation tools create a dictionary and then create a file that matches the dictionary specifications.

Creating and Modifying the Dictionary

You can add, alter or delete dictionary definitions as long as the data file does not exist.

When using Paint or the Define File window, you can add or delete field definitions at will, provided that you do not choose to create the data file.

Once the data file has been created, you will encounter restrictions on some types of Advanced Revelation dictionary modification:

- You may add or delete *only* type G (Group) or type S (Symbolic) fields or field synonyms.
- You cannot change the current *Dict type*, *Data type*, *Length*, *Value required* (Null/Not Null), and *Master definition* specifications for any column.

For more information, see “Modifying the File Structure” in this section.

About Record Keys

When you use Advanced Revelation tools to create a file you must specify at least one field that acts as the record key. When the table is created on the SQL Server, the bond automatically creates a unique clustered index on the record key column. For multipart keys, the bond creates a unique clustered composite index.

Defining Multipart Record Keys

In order to comply with restrictions on the SQL Server, your multipart key must obey these limits:

- Do not define more than eight key parts.
- The combined length of all columns in a multipart key cannot exceed 256 characters.
- Do not use a VARCHAR type column in a multipart key unless you limit the length. (The default length for the VARCHAR data type is 255 characters.)

Data Typing

There are some data types provided that are unique to the SQL Server Bond. These SQL Server types reflect their usage in the SQL Server environment. You can use these types or the standard Advanced Revelation data types.

The SQL Server bond-specific types are:

- MSSQL_BIT
- MSSQL_CHAR
- MSSQL_DATETIME
- MSSQL_INT
- MSSQL_SMALLINT
- MSSQL_TINYINT
- MSSQL_MONEY
- MSSQL_FLOAT
- MSSQL_VARCHAR
- MSSQL_TEXT
- MSSQL_BINARY
- MSSQL_VARBINARY
- MSSQL_IMAGE

If you use the standard Advanced Revelation data types, correspondences will be automatically applied to map to a type appropriate for the SQL Server bonding dictionary.

- ☛ See “Data Types” in “The SQL Server Bond Reference” section for bonding dictionary information and the standard type correspondences.

**Assigning Data Type
Attributes**

Whenever you specify a data type, two attributes are potentially available: length and decimal places. Their applicability is determined by the data type you choose. You will be prompted for the attributes when you enter the data type.

**Maintaining Field
Position**

You must maintain a contiguous sequence of fields in the dictionary. In other words, there must be a field defined for every position in the record.

- ☛ Every field has a *Position* number. You can display this information in the Define File window, in the Dictionary window, or by using the TCL command LISTDICT.

The bond will not allow you to create a file with missing fields. If you delete a field when defining the dictionary (before the data file exists), you will need to change the position numbers assigned to the remaining fields to reflect the new sequence.

Updating SQL Server Files

Using your SQL Server Bond, you can update SQL Server files as if they were Advanced Revelation files. At the same time, users of the SQL Server environment will see your changes as if they had been made in the SQL Server environment.

The following topics address issues of which you should be aware as you update SQL Server files.

Update Ability Controlled by the SQL Server The ability to update tables on the SQL Server is controlled by permissions granted on the server. In order to use Advanced Revelation tools to update, insert, or delete data in a table or a view on the server, you must have full access/update permission on the base table(s). If this is not the case, you must use Transact-SQL commands to make your updates.

Writing to Fields Any data you write to a field must conform to the type, length, and possibly decimal places of the field as defined in the Advanced Revelation dictionary. In addition, it may have to comply with an Advanced Revelation validation pattern and/or an SQL Server rule.

☛ For more information about domain validation and SQL Server rules, see “Advanced Revelation Validation Patterns and SQL Server Rules” in the “Reference” section in this chapter.

Loss of Length and Precision If you inadvertently violate a field definition by writing data that is too precise or too long by the SQL Server's definition, one of several results may occur:

- If there is a data validation rule on the SQL Server, it will be applied. If the rule is broken, an error message displays and the transaction is rolled back.
- If you are doing data entry in a window and there is an Advanced Revelation data validation pattern for the prompt, it will be applied. If data is invalid, an error message displays when the user presses enter at the prompt. The user must correct the entry before proceeding to the next prompt.

To protect against loss of data, you should establish domain validation on the SQL Server. To support the domain validation established on the SQL Server, you can use Advanced Revelation validation patterns in your data entry windows to catch invalid data as it is entered at a prompt.

Warning! If you do not use domain validation and the data entered is too long or too precise for the field, the write operation may succeed (it will not be rejected), but you may experience partial or total data loss.

The current SQL Server Bond does not allow you to specify acceptable degrees of data loss. The specifications at the *Acceptable Data Loss* prompts in the Dictionary Bonding Information window are ignored by the SQL Server Bond.

Adding a Record You can add records to a bonded SQL Server table or view at any time, provided you have full access/update permission on the base table(s).

Deleting a Record You can delete records from a bonded SQL Server table or view at any time, provided you have full access/update permission on the base table(s).

Copying Files and Records

You can use the usual commands in Advanced Revelation to copy and move data in the SQL Server bond. However, there are restrictions on doing so.

Copying Records In many cases, you can use the RECORDCOPY command to copy individual records to and from an SQL Server file. As a rule, the target environment determines whether you can copy a record to it. In the case of the SQL Server bond, the records you are copying to an SQL Server file must match the file structure. In other words, each field of each record must match the SQL Server definition for that field (data type, length, etc.). If you attempt to copy a record with invalid data to an SQL Server file, the copy operation will fail.

☞ If you are not using data validation on the SQL Server or in Advanced Revelation, the copy operation might be successful even if there is a mismatch. However, you may end up losing some data. For more information about potential data loss, see “Writing to Fields” above.

On the other hand, it is relatively easy to copy records from an SQL Server file into an Advanced Revelation file, since an Advanced Revelation file supports a much more flexible record structure.

Copying Files In the current release of the SQL Server Bond, you *cannot* use the TCL FILECOPY command to copy a table.

You can copy SQL Server tables using the SQL Server bulk copy utility. See your SQL Server system administrator documentation for information about this process. You can then attach the copied file from within Advanced Revelation, and the dictionary for it will be created automatically.

Modifying the File Structure

If you have appropriate permissions within the database, the SQL Server will allow you to explicitly modify an SQL Server table using the Transact-SQL command ALTER TABLE. In Advanced Revelation you can enter Transact-SQL commands at the SAW window.

When the structure of an SQL Server table changes, you must delete and recreate the existing Advanced Revelation dictionary for the table. See “Recreating the Dictionary” below for more information.

- ✎ Currently, you *cannot* use any Advanced Revelation tool to alter the structure of a table on the SQL Server.

Checking for Changes to the File Structure

A table on the server may be altered by processes outside of Advanced Revelation. For example, a user might execute a Transact-SQL ALTER TABLE command at the SAF window. When changes are made outside of Advanced Revelation, the dictionary will no longer accurately represent the current status of the table structure.

During an Advanced Revelation session, the SQL Server Bond checks for changes in a table’s structure the first time that you execute an Advanced Revelation process that opens the table. At that time, the bond compares the current table structure on the server to the existing dictionary.

The bond requires that all of the following things be true:

- An Advanced Revelation master dictionary record exists for every column defined on the SQL Server.
- Data type specifications in the Advanced Revelation dictionary and on the SQL Server match.
- Length specifications in the Advanced Revelation dictionary and on the SQL Server match.
- Null/Not Null specifications in the Advanced Revelation dictionary and on the SQL Server match.

If there is a discrepancy, the bond decides that the table structure has changed and displays an error message indicating that the dictionary is not properly synchronized.

The Advanced Revelation dictionary must always be synchronized with the table's definition in the SQL Server SYSCOLUMNS table. If it is not, the SQL Server Bond will not provide access to the table. Until the Advanced Revelation dictionary for the table accurately describes the table's current structure, you cannot perform SELECT, UPDATE, or DELETE functions against the table. The SQL Server Bond prohibits these functions in order to protect the integrity of the data.

Resolving Dictionary Synchronization Errors

The bond displays an error message if the existing dictionary and the current table structure are not synchronized. In this case, you cannot access the table until the dictionary is recreated so that it reflects the current table structure.

Recreating the Dictionary

Follow these steps to create a dictionary that reflects the current file structure:

1. If the current dictionary contains symbolic fields and synonyms, copy these to a temporary file.
2. At TCL enter the command:

```
DELETEDFILE DICT.tablename
```

to delete the current Advanced Revelation dictionary for the table.

Warning! If you do not include the designation "DICT.", the *data* portion of the table also will be deleted.

3. At TCL enter the command:

```
ATTACH volumename
```

to reattach the volume so that Advanced Revelation can recognize that changes have been made.

4. At TCL enter the command:

```
LISTDICT tablename
```

Because no dictionary exists, one will be generated automatically. The new dictionary will reflect the current structure of the table on the SQL Server.

5. After the new dictionary has been created, copy any symbolic and synonym fields back from the temporary file.

Modifying the Dictionary

If the table already exists on the SQL Server, the bond prohibits any attempts to add or delete master fields in the dictionary. In addition, the bond does not allow you to change the current *Dict type*, *Data type*, *Length*, *Value required* (Null/Not Null), and *Master definition* specifications for any column.

Using Existing SQL Server Views

Advanced Revelation views are used exclusively for viewing data and can only be accessed with an SQL SELECT statement. In contrast, SQL Server views can be updateable. If you update, insert, or delete data in an SQL Server view, your changes will be reflected in the base table(s) that define the view.

General Requirements for Accessing Views

You can access and modify SQL Server views from Advanced Revelation, with the following restrictions:

- The view and all base tables must be in the same SQL Server database.
- You can only modify one base table at a time. For example, if a view is based on three tables, and if you want to update every column in that view, you must perform three separate update operations.
- You cannot perform key-related operations, because views cannot have key columns. This includes accessing views with windows and accessing individual rows by key. (See “Valid Operations on Views” below for an exception.)

Building the Advanced Revelation Dictionary

The first time you use an Advanced Revelation tool to access an existing SQL Server view, an Advanced Revelation dictionary is dynamically created for the view. The existing columns in the SQL view are mapped to corresponding dictionary definitions in Advanced Revelation.

- ☞ For more information, see “Data types: Attaching SQL Server Tables” in the “Reference” section in this chapter.

The dictionary created for a view is assigned the standard Advanced Revelation identifier of:

`DICT.viewname`

where *viewname* is the name of the existing SQL Server view. To display the dictionary, use the TCL command LISTDICT.

Modifying Data in Views To modify data in views, you must:

- Use Advanced Revelation SQL INSERT, UPDATE, and DELETE statements.
- Include in the INSERT, UPDATE, or DELETE statement the key column(s) for the base table.
- Include values for all columns in the base table that are defined as NOT NULL. If a table contains three NOT NULL columns and only two of those columns appear in a view, you cannot update that table from the view.

For example, assume a CUSTOMERS table with a key column CUST_ID. Each salesperson has a view based on CUSTOMERS. Mary wants to update her view, MARY_CUSTOMERS, which will also update the base table, CUSTOMERS. She can only do so if:

- She uses an Advanced Revelation SQL UPDATE statement.
- CUST_ID is a column in the view MARY_CUSTOMERS.
- She includes a value for CUST_ID in the UPDATE statement.
- All NOT NULL columns in CUSTOMERS are included in MARY_CUSTOMERS.

Valid Operations on Views

As noted above, Advanced Revelation views can only be used to view data in base tables. You can only retrieve data using an Advanced Revelation SQL SELECT statement.

SQL Server views, on the other hand, can be treated much like base tables. The following Advanced Revelation operations can be executed against SQL Server views:

- Advanced Revelation SQL INSERT, UPDATE, DELETE, and SELECT statements.
- Embedded SQL (except UPDATE ... CURRENT OF and DELETE ... CURRENT OF).
- R/LIST reports.
- R/BASIC SELECT ... READNEXT loops.
- Queries.

SELECT ... READNEXT loops and Queries are key-related operations that would normally be invalid against a view. However, when you execute a SELECT ... READNEXT loop or a Query on a view, Advanced Revelation builds a temporary table, assigns arbitrary keys to the rows in that table, and reads values from the table.

- ✎ The temporary table is created in the current SQL Server database. You must have rights to the CREATE TABLE command in that database to execute a SELECT ... READNEXT.
- ✎ You cannot interrupt a SELECT ... READNEXT to call another recursive execution level and execute another SELECT ... READNEXT on the same SQL Server view.

Creating SQL Server Views

To create a view on the SQL Server, you must use the Transact-SQL command CREATE VIEW.

- ✎ In the current release of the SQL Server Bond, you *cannot* use the Advanced Revelation SQL CREATE VIEW command to create a view on the SQL Server. Similarly, you *cannot* use the Advanced Revelation DROP VIEW command to drop a view on the SQL Server.

Views that you create on the SQL Server are made up strictly of SQL Server tables. You cannot use Transact-SQL to create a view on the SQL Server that mixes Advanced Revelation files and SQL Server tables.

Creating a Multiple-Table Bonded View

You can use Advanced Revelation's SQL CREATE VIEW command to create a view that combines data from SQL Server tables, Advanced Revelation files, and other kinds of bonded files. Because it is an Advanced Revelation view, it will not be updateable.

Planning Security

Advanced Revelation and SQL Server both provide a variety of security features. Both have login name/password security. In addition, Advanced Revelation offers restriction levels to secure access to application windows within accounts. The SQL Server allows you to maintain security on database objects (tables, views, columns, and stored procedures) and on certain commands.

Accounts vs. Databases

The security measures that are available in an Advanced Revelation accounts are roughly comparable to the security measures that can be enforced in an SQL Server databases. Accounts and databases are both logical groups of files that contain related information. The main purpose of accounts and databases is to provide a basis for controlling access to data and commands.

Advanced Revelation enables you to restrict access to accounts and, within accounts, to windows. By creating windows that contain only the fields that you want users to see, you can control access to fields. You can also remove commands from an account's VOC file and restrict access to TCL commands.

- ☛ For more information, see the chapter "Users and Accounts: Planning Access and Security" in the section *Developing an Application* and the chapter "Introduction to TCL" in the section *TCL*.

On the SQL Server, you can use the commands GRANT and REVOKE to give or rescind access to tables, views, columns, stored procedures, and certain commands.

- ☛ See your SQL Server documentation for more information on SQL Server security.

Logins & Passwords

If any of the files (tables) in your application are located on the SQL Server, you will have to pass through SQL Server security as you attach these files. The Advanced Revelation volumes that you set up for the SQL Server contain login information that SQL Server security requires before permitting access to data on the server.

- ☛ For details about creating volume definitions, see "Setting Up Advanced Revelation Volumes for the SQL Server" in this chapter.

When you set up an SQL Server volume, login and password specifications are entered at the *Login Name* and *Password* prompts in the MS-SQL Volume Attributes window. There are three different ways to set up a volume definition to comply with SQL Server security:

- Include a login name and password in the volume definition.
- Specify that the SQL Server Bond should prompt the user for a login name and password the first time files in the volume are attached.
- Coordinate Advanced Revelation and SQL Server user/login names and passwords, and specify that the SQL Server Bond should pass the current login name and password to the SQL Server the first time files in the volume are attached.

When files in an SQL Server volume are attached in Advanced Revelation, the volume definition determines which method the bond will use to provide the login name and password to the SQL Server. Each of the three methods is described in more detail below.

Include a User Name and Password in the Volume Definition

A volume definition can include a login name and encrypted password information. If you do include an SQL Server login name and password, the login name and password are automatically passed to the SQL Server whenever you attach SQL Server tables. As long as the name and password in the volume definition correspond to a name and password on the SQL Server, you are allowed access without further intervention.

- ☛ This method requires a separate volume definition for each user who wants to access the specified database.

At the *Login Name* prompt, enter the login ID that you want the bond to use to log onto the SQL Server.

If there is an SQL Server password assigned to the specified login ID, enter that password at the *Password* prompt. A password is required only if the SQL system administrator assigned one for use with your log-in ID.

Warning! A password in the volume definition does not guarantee secure data on the SQL Server. To access someone else's data on the SQL Server, you need only know the name of that user's volume definition. If you enter "ATTACH *volume_name*" at the TCL window, the tables in that volume are attached and accessible.

If your SQL Server is defined as case sensitive, be careful to enter your login name and password precisely as they are defined on the SQL Server.

Ask the User for a User Name and Password

You may prefer to have the bond prompt users for an SQL Server login ID and password. To do so, enter "???" (double question mark) at the *Login Name* prompt, the *Password* prompt, or both. When you first attach tables on that volume, Advanced Revelation displays a window that asks for the necessary information.

- ☛ With this method, you need only one volume definition to provide all authorized users with access to the specified database. And you need not coordinate Advanced Revelation user name and password with your SQL Server login ID and password.

Coordinate Advanced Revelation and SQL Server User/Login Names and Passwords

If you coordinate the Advanced Revelation user name and password with the SQL Server login ID and password, you can omit the login name and password from the volume definition and avoid prompting users for those values. Instead, you can direct Advanced Revelation to pass the current Advanced Revelation user name and password directly to the server.

- With this method, you need only one volume definition to provide access to the specified database, but you must always keep Advanced Revelation and SQL Server user/login names and passwords synchronized.

Advanced Revelation always converts the Advanced Revelation user name and password to uppercase during the logon process. The name and password are passed to the server in uppercase. If your SQL Server is case sensitive, take care to create uppercase login IDs and passwords on the server.

To use this method, enter “%%” (double percent sign) at the *Login Name* prompt, the *Password* prompt, or both. When you first attach tables on that volume, the bond logs onto the SQL Server using the same name and/or password that was entered by the user when logging onto Advanced Revelation. If your Advanced Revelation user name and password correspond to a name and password on the SQL Server, you are allowed access without further intervention.

For example, Josh has logged into Advanced Revelation using his name and password. Because neither an SQL Server login name nor a password is included in the volume definition, Josh's Advanced Revelation user name and password are passed to the SQL Server when he attaches SQL Server tables. If either the user/login names or the passwords do not match, Advanced Revelation will return a message telling Josh that he has been denied access to the SQL Server.

SQL Server Database Security and Multiuser Issues

If the security of an SQL Server database is not of great concern, you can use one volume definition for all of your Advanced Revelation users. However, under certain conditions, you may need to create multiple volume definitions to accommodate multiuser access to a database:

- *If* you are including an SQL Server login name and password in the volume definition, *and*
- *if* you have more than one user accessing the same SQL Server tables, *and*
- *if* it is important to you that they log in under separate login names and passwords (for example, they have different levels of access on the SQL Server),

then you need a separate volume definition for each user. All volume definitions will be identical except for the volume definition name and the login name and password of the user.

For example, you might create volume definitions CONTRIBUTORS_JULIE and CONTRIBUTORS_BENJAMIN. Each volume definition accesses the same database on the SQL Server, but they contain different login names and passwords.

If you include an ATTACH command in each user's logon verb, Advanced Revelation will automatically attach the SQL files at logon. The logon verb for Julie would include the command:

```
ATTACH CONTRIBUTORS_JULIE
```

while Benjamin's would include the command:

```
ATTACH CONTRIBUTORS_BENJAMIN
```

SQL Server User Types and Permissions

Your right to access objects and execute commands on the SQL Server depends on two factors: your user type and the rights granted to you by other users.

SQL Server divides users into four types:

- The system administrator (SA).
- Database owners (DBO).
- Database object owners.
- Other users.

On the SQL Server, you can grant your rights (the rights based on your user type) to other users and be granted rights by other users. Advanced Revelation does not currently support security based on user type and the granting of rights.

However, all Advanced Revelation interactions with the SQL Server must pass through SQL Server security. You cannot access the SQL Server from Advanced Revelation without logging into the SQL Server as a specific user (see “Logins and Passwords” above). Therefore, whenever you access the SQL Server from Advanced Revelation, SQL Server automatically checks your rights before allowing you to access objects or execute commands.

SQL Server object and command permissions apply not only to Transact-SQL statements but also to corresponding Advanced Revelation processes. For example, if you grant someone the right to use the CREATE TABLE command on the SQL Server, that user can also create a table on the SQL Server by:

- Executing an Advanced Revelation SQL CREATE TABLE statement.
- Creating a window with Paint.
- Executing the TCL command DEFINEFILE.
- Accessing the Define File window on the **Tools-Files** menu.

Each of these processes is an analog to the SQL Server CREATE TABLE command.

Maintaining Object and Command Permissions

SQL Server's object and command security provides tremendous flexibility in controlling access to data. You may want to simplify security maintenance by keeping widely-used objects (application tables, for example) in the name of the database owner (DBO). The DBO can then grant and revoke permissions for the entire application.

Indexing

You can use SQL Server indexes with your bonded SQL Server tables.

Creating SQL Server Indexes with Transact-SQL

SQL Server indexes are located on and maintained by the SQL Server. Therefore, indexes that you create on the SQL Server must conform to all SQL Server restrictions and rules. A general overview of SQL Server indexing is provided below. For more information about creating indexes on the SQL Server, consult your SQL Server documentation.

Only the owner of a table may create an SQL Server index for it. This rule is enforced by the SQL Server. If you are the table owner, you can explicitly create SQL Server indexes using the Transact-SQL command CREATE INDEX.

- ☛ Within Advanced Revelation, Transact-SQL commands are entered at the SQL Server Access window (SAW). For information about SAW, see “Using the SQL Server Access Window” in this chapter. For details about the Transact-SQL CREATE INDEX command, see your SQL Server Transact-SQL documentation.

Using Transact-SQL, you can set up SQL Server indexes for individual columns in a table. You can also create *composite* indexes in which specified columns in a table are combined and treated as one index entry. Three keywords are used with the Transact-SQL CREATE INDEX command to assign specific characteristics to an index: CLUSTERED, NONCLUSTERED, and UNIQUE. An index is either *clustered* or it is *nonclustered* (the default). In addition, you can specify that the values in an index must be *unique*. That is, no two rows are permitted to have the same index value. Composite, unique, clustered, and nonclustered indexes are described below.

☞ SQL Server will not allow you to create an index for a column that uses the TEXT, IMAGE, or BIT data type. For more information, see “Data Types that Cannot Be Indexed” later in this section.

Composite Indexes Composite indexes allow you to search two or more columns as a unit. In a composite index, the values in all specified columns are combined and treated as one index entry. For example, an Address table might have a composite index for the Lastname and Firstname columns.

To create a composite index, your Transact-SQL CREATE INDEX command would list the columns to be included. The order of your list of columns determines the sort-order priority for the indexed values. In the example above, the composite index would be sorted by last name and then by first name.

Unique Indexes By using the UNIQUE keyword in your Transact-SQL CREATE INDEX command, you can prohibit duplicate values in the index. In effect, a unique index specifies that each row in the table must contain a unique value in the indexed column.

You can use the UNIQUE key word for both clustered and nonclustered indexes.

When to Index. Under most circumstances, for existing SQL Server tables that you will access in Advanced Revelation, you should set up a unique index on the primary key (the column(s) that uniquely identify the row. For more information about the relationship of indexing and primary keys, see “Accessing SQL Server Tables That Are Indexed” and “Controlling How the Bond Assigns Keys” in this section.

On the SQL Server, a unique index provides a data integrity check. For example, you might use a unique index on the Library_Card# in a Readers table, since every reader must have a different card number. The unique index on Library_Card# enforces this requirement and prevents errors in data entry.

Clustered Indexes

Use the CLUSTERED keyword in your Transact-SQL CREATE INDEX command to create a clustered index. A clustered index causes the rows in a table to be sorted so that their physical order is the same as their logical (indexed) order. As data is updated or inserted, the SQL Server indexes and sorts rows on an ongoing basis.

☞ There can be only one clustered index per table.

When to Index. Under most circumstances, for tables that you will access in Advanced Revelation, you should create a unique clustered index on the primary key (the column(s) that uniquely identify the row). For more information about the relationship of indexing and primary keys, see “Accessing SQL Server Tables That Are Indexed” and “Controlling How the Bond Assigns Keys” in this section.

The clustered index synchronizes both the physical and logical order of the data. Therefore, rows containing the values in a range are physically adjacent to one another. A clustered index is useful on a column that is often searched for ranges of values. Once the row with the first value in the range is found, successive values in the range are immediately available.

Nonclustered Indexes

Use the NONCLUSTERED keyword in your Transact-SQL CREATE INDEX command to create a nonclustered index. In a nonclustered index, the physical order of the rows is not the same as their indexed order. A nonclustered index maintains a sorted list of indexed values. Associated with each index value is a pointer to the row with that value. There can be up to 249 nonclustered indexes per table.

☞ If you do not use either the CLUSTERED or NONCLUSTERED keyword in your Transact-SQL CREATE INDEX command, a nonclustered index is created.

When to Index. Each nonclustered index provides access to the data in a different sorted order. The availability of indexes will speed up any process on the SQL Server that involves sorting a column. The index provides the sorted list immediately. If no index is available, the system has to look at each row one by one. (A nonclustered index will speed up non-ordered searches as well.)

Its a good idea to use a nonclustered index on any column that is often accessed in sorted order (that is, specified in the ORDER BY clause). For example, if you frequently query a Products table and ORDER BY Supplier, you can speed up queries by creating a nonclustered index on Supplier.

Columns that are frequently used in joins are also good candidates for nonclustered indexes. The SQL Server performs a join faster if these columns are in sorted order.

**Advanced Revelation
Processes that Create
SQL Server Indexes**

When you use Advanced Revelation file creation tools (Define File window, Paint, TCL DEFINEFILE command, Advanced Revelation SQL CREATE TABLE command, SQL Create Table window) to create a table on the SQL Server, the bond automatically creates a unique index on the SQL Server.

The Advanced Revelation file creation tools all create a dictionary and then create a file that matches the dictionary specifications. On the SQL Server, the bond automatically creates the following index types for a column, based on the Advanced Revelation dictionary information for that column:

Dictionary Characteristic	SQL Server Index Type
Key to the record	Unique clustered index
Part of multipart key	Unique composite clustered index

Warning! If you make a UNIQUE specification in your Advanced Revelation SQL CREATE TABLE statements or at the Dictionary Bonding Information window, it will be ignored by the SQL Server Bond. If you want to specify that a column is UNIQUE, use Transact-SQL to create a unique index directly on the SQL Server. You can issue Transact-SQL commands at the SAW window within Advanced Revelation.

**Indexing Limitation for
Multipart Record Keys**

For multipart keys, the bond creates a unique clustered composite index. However, on the SQL Server each entry in a clustered or composite index is limited to 256 characters total length. Therefore, the combined length of all columns in a multipart key can not exceed 256 characters.

- ✎ The default length for the VARCHAR data type is 255 characters. Consequently, you cannot use a VARCHAR type column in a multipart key unless you limit the length.

**Data Types that Cannot
Be Indexed**

SQL Server will not allow you to create an index for a column that uses the TEXT, IMAGE, or BIT data type. Therefore, for columns that you define in Advanced Revelation using the TEXT, IMAGE, or BIT data types:

- Do not use the UNIQUE specification in an Advanced Revelation SQL CREATE TABLE statement.
- At the Dictionary Bonding Information window and at the SQL Create Table window, do not specify that the column is unique.
- Do not use the KEY specification in an Advanced Revelation SQL CREATE TABLE statement.
- At the Dictionary window, the Define File window, and the SQL Create Table window, do not specify that the column is the record key or part of the record key.

**Accessing SQL Server
Tables That Are Indexed**

When you use the SQL Server Bond to access an existing table on the SQL Server, the bond automatically creates an Advanced Revelation dictionary for the table. If SQL Server indexes were set up for the table, the bond reviews the indexes and makes entries in the Advanced Revelation dictionary based on some of the indexing attributes.

In the Advanced Revelation dictionary, the bond assigns the following characteristics to a column, based on SQL Server indexing on that column:

SQL Server Indexing Type	Dictionary Characteristic
Unique clustered index	Key to the record
Unique composite clustered index key	Part of multipart key

- ✎ Existing unique indexes on the SQL Server are not recognized in the Advanced Revelation dictionary. At the Dictionary Bonding Information window, the *Force unique values* prompt is not set to yes.

How the Bond Assigns Keys

To designate key column(s), the bond determines what is the lowest numbered unique index in SYSINDEXES. The column(s) associated with that unique index is defined as the record key in the Advanced Revelation dictionary.

A unique clustered index is always the lowest numbered index in SYSINDEXES. So if a unique clustered index exists, the column(s) for that index is defined as the key. However, if there is no unique clustered index, the bond will designate a key column(s) based on the lowest numbered nonclustered unique index.

Controlling How the Bond Assigns Keys

If the bond will access an existing SQL Server table, you should create a unique clustered index for the column(s) that you want to use as the primary key(s) (subject to the SQL Server's 256 character length limitation on entries in clustered indexes).

The bond *does not* rely on information in the SYSKEYS table in order to determine which column(s) should be defined as the record key(s). If you use the stored procedure SP_PRIMARYKEY on a column, but you do not create a unique clustered index for the column, that column will not be recognized as the key within Advanced Revelation.

Accessing SQL Server Tables That Are Not Indexed

To be able to use all Advanced Revelation tools on an SQL Server table, there must be *at least one SQL Server unique index* assigned to a column in the table. The bond determines the key column for the table based on the SQL Server indexes set up for that table. See "How the Bond Assigns Keys" above for details.

If you do not have a unique index assigned to any of the columns in a table, the SQL Server Bond cannot recognize a key field. Therefore, you will not be able to access this table by any Advanced Revelation process that requires a key (for example, data entry window operations). You will only be able to work with the table using Transact-SQL and Advanced Revelation SQL commands.

Using Advanced Revelation Indexing on SQL Server Tables

The use of Advanced Revelation indexes is *not* encouraged. However, you may use Advanced Revelation indexes successfully under these conditions:

- You are *not* performing SQL INSERT, UPDATE, or DELETE functions on the indexed table.
- The SQL Server table will *not* be accessed or updated by processes outside of the Advanced Revelation environment.

When using the SQL Server Bond, note the following limitations:

- Adding Advanced Revelation indexes to your bonded SQL Server tables *is not recommended*. Both within and outside of Advanced Revelation, there are table update processes that may bypass the Advanced Revelation indexing system. If an update process bypasses the Advanced Revelation indexing system, the Advanced Revelation index will be out of date and will have to be rebuilt. There is no automatic way from within Advanced Revelation to detect whether an index will need to be rebuilt.
- The SQL Server Bond does not support Quickdex/Rightdex indexes. This index type would violate domain constraints of an SQL Server table because it adds a %RECORDS% field to the table, altering the table structure.
- The TCL INDEX command and the **Tools-Files-Indexing-Index Update** option are not supported by the SQL Server indexes. These update functions apply exclusively to Advanced Revelation indexes.
- The Advanced Revelation system subroutines BTREE.EXTRACT and COLLECT.IXVALS are not supported by the SQL Server indexes. These are strictly Advanced Revelation index (SI.MFS) access routines.

The SQL Server Bond Reference

The following information is important for your proper use of the SQL Server Bond:

Advanced Revelation SQL Commands and Keywords

The following Advanced Revelation SQL commands and keywords operate differently in the SQL Server environment than they do in Advanced Revelation:

- ALTER TABLE** Currently, you cannot alter the structure of a table on the SQL Server using the Advanced Revelation SQL ALTER TABLE command. Instead, you must use the Transact-SQL ALTER TABLE command from the Server Access Window and then recreate the Advanced Revelation dictionary. See “Modifying the File Structure” under the topic “Using the SQL Server Bond”.

- AVG** The Advanced Revelation SQL AVG keyword, which is used in a SELECT statement to calculate averages, returns different values when used within the Advanced Revelation environment than when used on data in SQL Server tables:
- When used on SQL Server tables, AVG returns an integer value, just as the Transact-SQL AVG keyword does.
 - When used on Advanced Revelation tables, AVG returns a decimal value.
- CREATE VIEW and DROP VIEW** The Advanced Revelation SQL commands CREATE VIEW and DROP VIEW operate exclusively on Advanced Revelation views. You cannot use Advanced Revelation SQL to create or drop a view on the SQL Server.
- ESCAPE** When searching for strings using the LIKE clause, do not use the ESCAPE clause to replace the default “\” with another character. Use of the ESCAPE clause with the SQL Server Bond has not been implemented.
- SET TRANSACTION** In transactions involving SQL Server tables, locking is pre-set to consistency level 3 (exclusive locking for the duration of the transaction during modifications and shared locking during queries). The SET TRANSACTION statement will not affect the locking of SQL Server tables.
- UNIQUE** The UNIQUE key word, used with the Advanced Revelation SQL CREATE TABLE command, is not supported in this version of the SQL Server Bond. If you use UNIQUE with the SQL CREATE TABLE command, or set *Unique* to yes at the SQL Create Table window or at the Dictionary Bonding Information window, your specification is ignored. If you want to specify that a column is unique, use Transact-SQL to create a unique index directly on the SQL Server. You can issue Transact-SQL commands at the SAW window within Advanced Revelation.
- SQL Cursor Commands** The current release of the SQL Server Bond does not support Advanced Revelation SQL cursors. Therefore, the following embedded SQL commands may not be used:

```
DECLARE CURSOR
OPEN
FETCH
UPDATE CURRENT OF
```

Instead, you can simulate cursors by using the embedded SQL SELECT, which can return multiple rows. See SELECT (Embedded SQL) in *SQL* for more information and examples.

Warning! The use of Advanced Revelation SQL cursors yields unpredictable results.

R/BASIC Programs

In general, R/BASIC programs that access data on the SQL Server are the same as programs that access data within Advanced Revelation. Make note of the following exceptions.

- SELECT/READNEXT Loops** If you select record keys with an R/BASIC SELECT or SELECT BY statement or an embedded SQL SELECT statement and then read through the records with a READNEXT command contained in a loop, you must either:
- Read to the end of the select list with READNEXT, or
 - Clear the select list with a CLEARSELECT as soon as you exit the loop.

Warning! Uncleared select lists will either occupy connections to the SQL Server or clutter your system with temporary tables. If you have too many open connections, you may encounter memory corruption.

Naming Conventions

When working with SQL Server tables, Advanced Revelation and the SQL Server follow specific naming conventions. The use of names is explained below.

Naming Tables and Views

When you create tables and views on the SQL Server from Advanced Revelation, the table, view, and column names must meet certain requirements. For the sake of portability, tables and views in Advanced Revelation should meet the same requirements:

- 30 characters maximum. (The current ANSI standard specifies 18 characters.)
- The first character must be upper case alphabetic (A-Z) or “_” (underscore). (Advanced Revelation does not allow you to create or access SQL Server temporary tables, which begin with a “#” (pound) symbol.)
- All other characters must be alphanumeric (A-Z, 0-9) or the symbols “\$” (dollar sign), or “_” (underscore).
- Names cannot contain a “.” (period). SQL Server objects are qualified by adding a database name and a user name. These qualifiers are separated from the object name with periods. See “SQL Server Object Names”.

- Names cannot contain Advanced Revelation SQL or Transact-SQL commands or keywords.

Ambiguous Table and View Names

If your SQL Server is case sensitive, SQL Server table and view names may contain lower case alphabetic characters (a-z) and “#” symbols after the first character. However, you cannot create SQL Server tables and views from Advanced Revelation with these characters in the name except from the Server Access Window.

When you attach SQL Server tables and views whose names include “#” and the lower case alphabet, Advanced Revelation converts “#” symbols to “_” and the lower case alphabet to upper case. For example, all of these valid SQL Server names:

```
NEW#CUSTOMERS
NEW_CUSTOMERS
new#customers
new_customers
New_Customers
```

convert to:

```
NEW_CUSTOMERS
```

in Advanced Revelation. In addition, a table with that name *may already exist* in Advanced Revelation.

As long as tables with identical names are owned by different users, you can attach and access them simultaneously. Simply address the tables by their fully-qualified names (for example, using the name format *owner.table_name*, you could attach JOHN.COMPANY_CONTACTS and LYDIA.COM-PANY_CONTACTS). If tables are not qualified with different user names, the last table attached is the only table available.

☞ See “Accessing Tables by Fully-Qualified Name” below.

SQL Server Object Names

All objects on the SQL Server (tables, views, stored procedures, etc.) use qualifiers in the same way to uniquely identify the object. This is true whether the objects are created with Advanced Revelation processes (the Define File window, for example) or with Transact-SQL statements (for example, CREATE TABLE). Qualifiers are listed in the order shown below. Each qualifier is separated from the next using a period:

```
database.owner.object_name
```

When you create an object on the SQL Server, the SQL Server automatically associates the name of the database and your user name with the object name. The fully-qualified name of a PROSPECTS table created by Julie on the SALES database is:

SALES.JULIE.PROSPECTS

Advanced Revelation Table Names

In anticipation of a future implementation of user-oriented security, some Advanced Revelation processes now add the user names to qualify Advanced Revelation table names. This topic explains when user names are added to table names, how to list these fully-qualified table names, and when you need to use them.

Attaching an Existing SQL Server Table

When you attach a table on the SQL Server to Advanced Revelation for the first time, Advanced Revelation creates and saves a name for that table. Under some circumstances, this Advanced Revelation table name includes a qualifier. Whether a qualifier is added to the name depends on who created the table on the SQL Server:

- If the database owner (DBO) created the table, Advanced Revelation's name for that table is simply the unqualified SQL Server table name. A table created by the DBO is assumed to be global, so no qualifier is added. Assuming that Julie is the DBO, the Advanced Revelation name of the SQL Server table SALES.JULIE.PROSPECTS is:

PROSPECTS

- If someone other than the DBO created the table, Advanced Revelation adds the creator's SQL Server *login* name to the table name, separated by an "@" symbol. Assuming that Julie is *not* the DBO, the Advanced Revelation name of the SQL Server table SALES.JULIE.PROSPECTS is:

JULIE@PROSPECTS

Remember, Advanced Revelation uses the creator's SQL Server *login* name to qualify a table name, while the SQL Server uses the creator's *user* name as a qualifier. Therefore, if the creator of an SQL Server table has an SQL Server login name and an SQL Server user name that do not match, the Advanced Revelation and SQL Server qualifiers for that table will not match.

Creating Tables on the SQL Server with CREATE TABLE

Generally, when you create a table using the Advanced Revelation SQL CREATE TABLE statement, Advanced Revelation adds the current user name (@USERNAME) to qualify the Advanced Revelation table name. The one exception is when you log into the SYSPROG account without a user name. In that case, no user name is added under the assumption that tables created from SYSPROG are global and therefore do not need a qualifier.

Advanced Revelation separates the user name from table name with an "@" symbol:

user_name@table_name

- ☛ See "Accessing Tables by Fully-Qualified Name" below for information on addressing tables by fully-qualified name.

Your user name is a function of your logon command. At log on, if you enter:

- A user name (with or without an account name), that is your user name.
- An account name only, your user name is the account name.
- Neither an account nor a user, your user name is either USER or COLOR.USER, depending on your video card.

Accessing Tables by Fully-Qualified Name

If you are logged onto Advanced Revelation with one user name and you want to access data in a table that belongs to another user, you must specify the fully-qualified name for that table.

When accessing a table by its fully-qualified name from Advanced Revelation SQL, separate the qualifier from the table name with a "." (period):

JULIE.PROSPECTS

When using any other Advanced Revelation process to access a table, separate the qualifier from the table name with an "@" symbol:

JULIE@PROSPECTS

Record Structure

Information about record structure in Advanced Revelation and on the SQL Server is provided below.

Row (record) keys

Note the following SQL Server limitations:

- SQL Server limits keys to 8 parts.
- SQL Server prohibits the use of MSSQL_BIT, BINARY, TEXT, or IMAGE data types for key columns. Because the Advanced Revelation generic VARBINARY data type maps to an IMAGE data type on the SQL Server, you cannot use a VARBINARY column as the key.

When you define key columns for an SQL Server table from within Advanced Revelation, the bond creates an SQL unique clustered index for the key column(s). Because the SQL Server restricts indexes to 256 characters, the length of the key cannot exceed 256 characters.

In addition, the bond activates the SQL Server system stored procedure **sp_primarykey**.

- ✦ For more information on the relationship between record keys and SQL Server indexes, see “Indexing” in this chapter.

The rules for accessing an SQL Server table with no primary key column are the same as the rules for accessing SQL Server views. See “Using Existing SQL Server Views” for more information.

Multivalued Fields

You can save and access multivalued data in an SQL Server table just as you can in an Advanced Revelation file, if:

- No users will ever access that data from any other environment, including the SQL Server, because no other environment properly interprets multi-values.
- You save the data in columns with data types of CHAR, VARCHAR, or TEXT.

If you are storing multivalued data in a CHAR or VARCHAR column, remember that a VARCHAR column on the SQL Server has a maximum length of 255 characters. If you enter too many values, data will be truncated without warning. You can use an Advanced Revelation validation pattern or a SQL Server rule to ensure that data is not truncated. For more information, see “Truncation Handling” in this section.

If you are storing multivalued data in a TEXT column, remember that the maximum length of an Advanced Revelation row/record is 64K.

- Do not attempt to save multivalued data in columns with data types other than CHAR, VARCHAR, or TEXT. Advanced Revelation separates values in a multivalued field with @VM (ASCII 253), which is an invalid character for all numeric data types.

Row and Column Maximum Lengths

The SQL Server is considerably more restrictive than Advanced Revelation in terms of record (row) length and maximum number of columns.

- Row length** The maximum row length in an SQL Server table is 1,920 bytes plus TEXT and IMAGE columns. Each TEXT or IMAGE column requires 16 bytes of the 1,920 byte maximum.
 - Although it is possible to have up to 119 TEXT and/or IMAGE columns in a table, you can have only one IMAGE column in any table that is accessed by Advanced Revelation.

Number of columns The maximum number of columns in an SQL Server table is 255.

Data Types: Attaching SQL Server Tables

When you attach an SQL Server table, Advanced Revelation generates a dictionary for the table. Data types in the Advanced Revelation dictionary are based on data types in the SQL Server table:

Table M4-2

Data Types: SQL Server to Advanced Revelation

SQL Server data type	Advanced Revelation data type
INT	INTEGER(-2147483648,2147483647)
SMALLINT	INTEGER(-32768,32767)
TINYINT	INTEGER(0,255)
FLOAT	FLOAT
CHAR	CHAR
VARCHAR	VARCHAR
TEXT	TEXT
IMAGE	VARBINARY
BINARY	VARBINARY (continued)

Table M4-2

Data Types: SQL Server to Advanced Revelation (continued)

SQL Server data type	Advanced Revelation data type
VARBINARY	VARBINARY
BIT	BOOLEAN
MONEY	DOLLARS
DATETIME	DATETIME
TIMESTAMP	N/S*
User-defined	**

* N/S = not supported (see "TIMESTAMP" below)

** See "User-Defined Data Types" below

In most cases, Advanced Revelation data types interact seamlessly with SQL Server data types. However, you should be aware of a few anomalies:

MONEY to DOLLARS The SQL Server MONEY data type always formats data with four digits after the decimal. In Advanced Revelation, you can choose to display zero to four digits after the decimal (MD0 to MD4). The default value is MD24.

To change the number of decimal places displayed in reports, simply change the *Output Format* prompt at the Dictionary window (**Tools-Files-Dictionary**).

BIT to BOOLEAN The SQL Server BIT data type does not accept null values. Therefore, when the dictionary is built, the *Value Required* prompt at the Dictionary Bonding Information window is set to "Yes" automatically.

BINARY, VARBINARY, and IMAGE to VARBINARY The SQL Server BINARY, VARBINARY, and IMAGE data types can contain ASCII characters 249 to 255, which are reserved characters in Advanced Revelation. When you access SQL Server data in BINARY and VARBINARY columns, Advanced Revelation automatically converts these characters using the VB conversion. To display data in its original format, use the VB output conversion.

Warning! Do not attempt to store a column (field) that contains ASCII characters 249 to 255 or you will corrupt the row/record.

To use the SQL Server IMAGE data type successfully, you must comply with these restrictions:

- There can be only one IMAGE column per table.
- The IMAGE column must be the last column in the record.

IMAGE data can and probably will contain Advanced Revelation system delimiters such as field marks. To ensure that Advanced Revelation can read the data in an IMAGE field and treat it as a single unit, the field must appear at the end of the record. If an IMAGE field appears in any other position in the record, Advanced Revelation may be unable to recognize where the IMAGE field ends and where the next field in the record begins.

When working with an IMAGE type field in an R/BASIC program, you should not use the EXTRACT function to access the data. Instead, use the FIELD statement.

TEXT to TEXT and IMAGE to VARBINARY

SQL Server TEXT and IMAGE columns can contain up to 2GB of data, but Advanced Revelation can only process rows up to 64K in length.

☞ See “Truncation Handling” in this section for more information.

DATETIME to DATETIME

The SQL Server only calculates dates in the range January 1, 1753 to December 31, 9999, while Advanced Revelation calculates dates from 5,879,610 B.C. to 5,879,610 A.D. Do not attempt to store dates on the SQL Server that fall outside the prescribed range.

TIMESTAMP

If the SQL Server table contains a TIMESTAMP column, Advanced Revelation simply ignores it when building the dictionary. You cannot read or display the timestamp for SQL Server rows.

User-Defined Data Types SQL Server user-defined data types are based on SQL Server system data types. When you attach a table that contains columns with user-defined data types, the bond maps the system data type for each column to the corresponding Advanced Revelation data type. In addition, the bond maps length and NULL or NOT NULL characteristics, if any, to the related Advanced Revelation values.

Data types: Creating SQL Server Tables from Advanced Revelation

When you create an SQL Server table from Advanced Revelation, you can use Advanced Revelation generic data types. These data types are mapped to SQL Server data types as indicated below.

You can also use the Advanced Revelation SQL Server-specific data types, which all begin with "MSSQL_". The chief reason to use SQL Server-specific data types is to provide you with more precise control over the format of your data. The Advanced Revelation generic data type INTEGER is mapped to the SQL Server data type INT, which is the largest integer field on the SQL Server. If you do not need that much precision, you can substitute the data types MSSQL_SMALLINT or MSSQL_TINYINT.

The disadvantage to using SQL Server-specific data types is that the data is no longer portable to other environments. dBASE, for example, cannot interpret "MSSQL_" data types, but can interpret the generic Advanced Revelation data types.

Table M4-3
Data Types: Advanced Revelation to SQL Server

Advanced Revelation generic data type	SQL Server data type
BOOLEAN	BIT
CHAR	CHAR
TIME	DATETIME
DATE	DATETIME
DATETIME	DATETIME
INTEGER	INT
DECIMAL	FLOAT
DOLLARS	MONEY
FLOAT	FLOAT
	(continued)

Table M4-3

Data Types: Advanced Revelation to SQL Server *(continued)*

Advanced Revelation generic data type	SQL Server data type
VARCHAR	VARCHAR
TEXT	TEXT
VARBINARY	IMAGE

In most cases, Advanced Revelation data types interact seamlessly with SQL Server data types. However, you should be aware of a few anomalies:

BOOLEAN to BIT See “BIT to BOOLEAN” above.

The SQL Server BIT data type cannot be indexed, so it cannot be used as part of a key. In addition, the BIT data type does not accept null values. Therefore, when you use the BOOLEAN data type, the *Value Required* prompt at the Dictionary Bonding Information window is set to “Yes” automatically when the dictionary field is saved.

CHAR to CHAR If you create a column on the SQL Server with a data type of CHAR, and if you define the column to allow nulls, the SQL Server will treat the column as if it has a data type of VARCHAR. The SQL Server takes longer to access VARCHAR data than CHAR data, so you should define columns that have a data type of CHAR to not allow nulls whenever possible.

☛ When creating a table on the SQL Server using the Define File window, TCL DEFINEFILE command, or Paint, note that Advanced Revelation allows nulls by default. If you want to define a column as not allowing nulls, you must change this default value at the Dictionary Bonding Information window before creating the table.

DOLLARS to MONEY See “MONEY to DOLLARS” above.

VARBINARY to IMAGE To use the Advanced Revelation VARBINARY data type successfully, you must comply with these restrictions:

- There can be only one VARBINARY column per table.
- The VARBINARY column must be the last column in the record.

See “BINARY, VARBINARY, and IMAGE to VARBINARY” above.

**TEXT to TEXT and
VARBINARY to IMAGE**

See “TEXT to TEXT and IMAGE to VARBINARY” above.
Also see “Truncation Handling” in this section.

**TIME and DATE to
DATETIME**

SQL Server has only a DATETIME data type. When you write either a time or a date to the SQL Server, Advanced Revelation fills in the unspecified value:

- For TIME columns, the date is filled with December 31, 1967. (The Advanced Revelation internal system date is the number of days since December 31, 1967.)
- For DATE columns, the time is filled with midnight. (The Advanced Revelation internal system time is the number of seconds after midnight.)

If you access the data from the Server Access Window or from another environment, you will see a full date and time even though the column in Advanced Revelation is either a DATE or a TIME.

See also “DATETIME to DATETIME” above.

Truncation Handling

Advanced Revelation can only access rows up to 64K long. Truncation handling has not been implemented in the SQL Server bond. The bond ignores the specifications at the *Acceptable Data Loss* prompts in the Dictionary Bonding Information window.

Warning! If you access a row on the SQL Server that is longer than 64K, Advanced Revelation will automatically truncate the row to 64K. When you write the row back to the SQL Server, the truncated value will be written over the existing value on the SQL Server, and you *will* lose data.

- ☛ Each SQL Server TEXT or IMAGE column can contain up to 2GB of data. Therefore, you should be particularly careful when accessing a SQL Server table that contains an IMAGE or TEXT type column.

The SQL Server also will automatically, and without warning, truncate data longer than the length defined for a column. For example, if you have defined a column as data type CHAR(3) and attempt to update that column with the value 1234, the SQL Server will automatically truncate the value when it is saved in the database. If you select the updated row from the database, the value returned for that column will be 123.

You can define Advanced Revelation validation patterns or SQL Server rules to ensure that you do not attempt to save invalid data to the SQL Server. You can also use validation patterns or rules for columns with fixed lengths to ensure that the SQL Server will not truncate data without warning.

Advanced Revelation Validation Patterns and SQL Server Rules

Advanced Revelation and the SQL Server both have means of ensuring the validity of data by checking data entry and modifications against a set of rules. For instance, you can verify that someone has entered exactly ten numbers in a phone number column (field) and not nine numbers or a street address. In Advanced Revelation this data checking is controlled by validation patterns and, on the SQL Server, by rules.

Maintaining Validation Patterns and Rules

Between Advanced Revelation and the SQL Server, there are three ways to assign validation patterns:

- In prompts in Advanced Revelation data entry windows.
- In the Advanced Revelation data dictionary.
- On the SQL Server, as rules that are bound to columns in the table.

You may well want to maintain validation patterns in all three locations:

- Prompt validation patterns are verified immediately. Therefore, you will not be surprised when you attempt to store the record on the SQL Server and receive an “invalid data” error message (assuming that all validation patterns match).
- Advanced Revelation dictionary validation patterns verify data when you execute an Advanced Revelation SQL INSERT or UPDATE statement, or some R/BASIC programs, while prompt validation patterns do not.
- SQL Server rules fill in values for Transact-SQL statements and for Advanced Revelation processes that are not covered by other validation patterns (for example, some R/BASIC programs and the Editor).

Synchronizing Validation Patterns and Rules

If you do not synchronize Advanced Revelation validation patterns with SQL Server rules, you may be faced with two problems:

- The SQL Server may reject data that an Advanced Revelation process accepted.
- Data in the table may be inconsistent. As an example, assume a validation pattern on a phone number prompt that requires *exactly* ten digits and a rule that requires *up to* ten digits. With a Transact-SQL INSERT statement, you could enter a seven-digit phone number (without the area code), which the data entry window would reject.

Warning! If Advanced Revelation prompt and dictionary validation patterns and SQL Server rules do not have the same restrictions, you may be allowed to enter inconsistent data in the table.

There is no process for automatically synchronizing validation patterns and rules. You must set the window prompt, dictionary, and SQL Server values individually.

- Set validation patterns for columns in Advanced Revelation tables at the Dictionary window.
- See the chapter “Adding Advanced Features to Windows with Paint” in the section *Developing an Application* for information on defining prompt validation patterns.
- See the SQL Server documentation for information on creating rules for columns in SQL Server tables. Remember that you can perform SQL Server processes at the Advanced Revelation Server Access Window.

Default Column Values

Advanced Revelation and the SQL Server both have methods for automatically assigning a value to a column (field) if you do not enter a value for that column. You can assign a default value:

- To a prompt in a data entry window. If you leave that prompt blank when entering data, the prompt fills in the default value for you.
- To a column when you create a table, either in Advanced Revelation or on the SQL server. When you save a record to the table, if you have left that column blank, the column fills in the default value.

Warning! If you make a DEFAULT specification in your Advanced Revelation CREATE TABLE statements or at the Dictionary Bonding Information window, it will be ignored by the SQL Server Bond. If you want to specify a column default, use Transact-SQL to create a default directly on the SQL Server.

An SQL Server default can be activated by a Transact-SQL statement, any Advanced Revelation process, or another database manager.

Maintaining Defaults Advanced Revelation and the SQL Server have three ways to assign default values:

- At prompts in Advanced Revelation data entry windows.
- At the Dictionary Bonding Information window ([Shift-F5] from the Dictionary window).
- On the SQL Server, as defaults that are bound to columns in the table.

You may well want to maintain defaults in all three locations:

- Prompt defaults are visible during data entry. Therefore, you will not be surprised later when you run a report and find values that you did not enter.
- Advanced Revelation dictionary defaults fill in values when you execute an Advanced Revelation SQL INSERT or UPDATE statement, while prompt defaults do not.
- SQL Server defaults fill in values for Transact-SQL statements and for Advanced Revelation processes that are not covered by other defaults (for example, R/BASIC programs and the Editor).

Synchronizing Defaults Prompt defaults, column defaults in the Advanced Revelation dictionary, and defaults on the SQL Server must all be synchronized, or the default values will be inconsistent.

Warning! If your window prompt, Advanced Revelation dictionary, and SQL Server default values do not match, different processes will produce different default values.

If you do not synchronize defaults, the window prompt default value will prevail, because the prompt fills in the default value before sending the row (record) to the table. At that point the column has a value, so the column default is not used.

There is no process for automatically synchronizing defaults. You must set the window prompt, dictionary, and SQL Server default values individually.

- Set default values for columns in Advanced Revelation tables at the Dictionary Bonding Information window ([Shift-F5] at the Dictionary window or [Shift-F4] at the Define File window).
- See the chapter “Adding Advanced Features to Windows with Paint” in *Developing an Application* for information on defining prompt defaults.
- See the SQL Server documentation for information on creating default values for columns in SQL Server tables. Remember that you can perform SQL Server processes from the Advanced Revelation Server Access Window.

SQL Server Triggers

Triggers are SQL Server devices for cascading changes through a database. If you modify data in an SQL Server table and that table has a trigger bound to it, your changes can automatically cause changes to data in other SQL Server tables.

For example, if you add a row to an invoices table and you want that insertion to be reflected in a customer table, you can create a trigger that will automatically modify the customer table.

Each table can have up to three triggers, one each for UPDATE, INSERT, and DELETE statements. Triggers can also be combined, so you might have one trigger that is tripped by both UPDATE and INSERT statements.

Advanced Revelation and Triggers

Like Transact-SQL statements, any Advanced Revelation operation that modifies rows will trip a trigger. If you add a row to an SQL Server table with a data entry window, the bond translates that process into an SQL Server INSERT statement. If there is an INSERT trigger for that table, the trigger is tripped, and the trigger processes are executed.

Whether you change data in an SQL Server table using an Advanced Revelation data entry window, an R/BASIC program, or an SQL statement, your change will trip the trigger for that table, and the trigger will produce the appropriate changes.

Creating Triggers from Advanced Revelation

You can create triggers for SQL Server tables from the Server Access Window. Remember that triggers are a function of the SQL Server, so they can only modify tables in the SQL Server environment. See your SQL Server documentation for more information.

SQL Server Stored Procedures

The SQL Server enables you to assemble and save groups of complex or frequently-used Transact-SQL statements as “stored procedures” for later execution. In addition, the SQL Server has a host of pre-defined system stored procedures, which perform standard system administration functions.

- User-defined** A user-defined stored procedure consists of one or more Transact-SQL statements. You might regularly execute a complex join statement or an update that changes multiple tables and, therefore, requires multiple statements. By putting these statements into a stored procedure, you save having to re-enter them time and again.
- System** Access to SQL Server system administration tables is carefully regulated, and many system administration tasks can only be executed with system stored procedures. System stored procedures are pre-defined processes that enable you to alter and report on system tables, which contain information on security, data definition, and system administration.
 - ☛ Stored procedures operate only on the SQL Server environment. You can create user-defined stored procedures and run both user-defined and system stored procedures from the Advanced Revelation Server Access Window, but these procedures will only affect SQL Server data.

Creating and Executing Stored Procedures

You can create and execute stored procedures from the Advanced Revelation Server Access Window. Enter the command just as you would in the SQL Server SAF window. For example, execute a stored procedure by entering:

```
execute procedure_name
```

If the procedure is the first or only procedure in a batch, enter:

```
procedure_name
```

Stored procedures can be executed from R/BASIC code by directly calling the MSSQL_C routines. See the “MSSQL_C Routine” section for more information.

You cannot access Advanced Revelation files with stored procedures.

Null Handling

Advanced Revelation does not distinguish between a null value and a zero length string. Both are stored as a zero length string.

The SQL Server, on the other hand, does distinguish between a null value and a zero length string. An SQL Server null value indicates that no value was assigned to the column. A null value is stored on the SQL Server as a zero length string. A string comprised of zero or more spaces is stored on the SQL Server as 1 space.

The differing treatments of nulls and zero length strings in the two environments have some effects that are worthy of note.

Updating Data on the SQL Server

When saving data to a row (record), Advanced Revelation eliminates trailing spaces in all columns. Therefore, one space, which is the way a string of zero or more spaces is stored on the SQL Server, becomes a null value. This means that you cannot save a zero-length string on the SQL Server from Advanced Revelation.

Warning! If you modify a row in an SQL Server table using any Advanced Revelation process (except an SQL UPDATE statement see below), and if that row contains zero-length strings, you will lose data. When you save the row, Advanced Revelation rewrites the entire row, including any columns that you did not modify. All zero-length strings are replaced with null values.

An SQL UPDATE statement is the only Advanced Revelation process that does not replace *every* SQL Server zero-length string in a row with a null value. The bond sends UPDATE statements directly to the SQL Server, and the SQL Server updates only the columns specified in the statement. However, the bond still eliminates trailing spaces (including SQL Server zero-length strings) from the values being stored.

Comparisons Using Null Advanced Revelation SQL INSERT, UPDATE, DELETE, and SELECT statements that include:

```
... WHERE col_name IS NULL ...
```

or:

```
... WHERE col_name IS NOT NULL ...
```

search not only for NULL or NOT NULL values but also for zero length strings or non-zero length strings. The bond automatically converts these statements to:

```
... WHERE col_name IS NULL OR col_name = ' ' ...
```

or:

```
... WHERE col_name IS NOT NULL AND col_name <> ' ' ...
```

As an example, assume a ZIP_CODE column in a CUSTOMERS table on the SQL Server. This ZIP_CODE column contains both null values and zero length strings. When you execute an Advanced Revelation SQL statement that includes the clause:

```
... WHERE ZIP_CODE IS NULL ...
```

each row that contains either a null or a zero length string in the ZIP_CODE column is returned.

To locate both null values and zero-length strings on the SQL Server using Transact-SQL, you must explicitly include the clause:

```
... WHERE col_name IS NULL OR col_name = ' ' ...
```

**Comparisons Using
= (equal to) and (not
equal to)**

As with IS NULL and IS NOT NULL, Advanced Revelation SQL INSERT, UPDATE, DELETE, and SELECT statements that include:

```
... column_name = '' ...
```

or:

```
... column_name <> '' ...
```

search for both null values and zero length strings.

Running R/LIST Reports R/LIST also searches for both null values and zero-length strings. The clauses:

```
... WITH column_name = ' ' ...
```

and:

```
... WITH column_name <> ' ' ...
```

will retrieve both null and zero-length string values from the SQL Server.

Locking

The SQL Server will perform locking for all users accessing data on the SQL Server. Locking is managed by the SQL Server automatically.

For all R/BASIC LOCK statements and system applications (for example, EDIT, window locking options), Advanced Revelation will use the default lock service — maintaining Advanced Revelation locks by using the file server's logical locking. Advanced Revelation will use the locking scheme provided by your network.

Consistency Levels In transactions involving SQL Server tables, locking is pre-set to consistency level 3 (exclusive locks for the duration of the transaction during modifications and shared locks during queries). The Advanced Revelation SQL SET TRANSACTION statement will not affect the locking of SQL Server tables.

Deadlock Detection and Handling All deadlock detection and handling is managed by the SQL Server.

The Lock Wait Process Use the *Lock Timeout* prompt at the Concurrency Control Environment window to specify the amount of time a workstation will wait for a lock. See "Implicit Locking" in the "Locking" chapter in *Networks*.

Your *Lock Timeout* specification is passed to the SQL Server. The SQL Server abides by your time setting.

Transaction Processing

Transaction processing is handled by Advanced Revelation, or the SQL Server, or by both environments:

- Advanced Revelation manages transaction processing for all transactions that do not include SQL Server tables. In the event of a system failure, use the recovery methods provided by Advanced Revelation.
- If a transaction involves only SQL Server table, the transaction will use the SQL Server's transaction processing service to commit changes to SQL Server tables. In the event of a system failure, use the recovery methods provided by the SQL Server.
- If you are using multiple SQL Servers in one transaction, the SQL Server two-phase commit process monitors the transaction. In the event of a system failure, use the recovery methods provided by the SQL Server.
- For transactions involving both SQL Server tables and Advanced Revelation files, both transaction processing services are used. The two-phase commit utility is used on the SQL Server. In the event of a system failure you must use the recovery methods available on the SQL Server and in Advanced Revelation. See "Coordinating System Recovery" below for more information.

Using the Transaction Status Window

Once a transaction is begun, you can see which transaction processing services will be used by checking the Transaction Status window.

To display the Transaction Status window, at TCL enter:

```
TRANSACTION STATUS
```

The Transaction Status window displays information about the status of transaction processing at your workstation. Information includes:

- A list of all files modified by the current transaction that are under control of the Advanced Revelation transaction control system.
 - ☞ Files accessed through a server bond are not listed at the *Affected Files* prompt.
- Server bonds participating in the current transaction. A server bond provides access to an environment that usually has concurrency controls, transaction processing, and locking of its own. Files accessed through the SQL Server Bond are under control of the server's two-phase commit transaction control system. Advanced Revelation's transaction control system will coordinate with all server bonds to ensure successful transaction processing.

Coordinating System Recovery

During the two-phase commit process, transactions are committed on the SQL Server first and then transactions are committed to Advanced Revelation files. If a system failure occurs during the commit phase you must coordinate recovery activities on the SQL Server and in Advanced Revelation.

- There will be information pending in the Advanced Revelation Commit Log. When you restart the system, a message will inform you that there is a transaction pending. You *must* “retry” the commit to complete transaction processing. Completing the commit phase will affect *only* the Advanced Revelation files involved in this transaction.
- The SQL Server will recognize that there is a pending transaction involving SQL Server tables. Use the SQL Server transaction recovery process to commit the transaction. Completing the commit phase will affect *only* the SQL Server tables involved in this transaction.

Warning! To ensure that your transactions are recovered, you must recommit on *both* the SQL Server and in Advanced Revelation. When a system failure occurs, it breaks the transaction processing tie between Advanced Revelation and the SQL Server. Neither environment will have any knowledge that there is a relationship between the transaction pending in their own environment and one in the foreign environment. It is up to you to ensure that all files involved in a transaction are handled in the same manner, regardless of their environment.

Error Handling

If an Advanced Revelation process results in an error on the SQL Server, there are a number of ways to determine the cause of the problem.

Error Messages If you attempt to execute an invalid operation on the SQL Server, an error message displays that explains where you went awry. Some SQL Server error messages have exact parallels in Advanced Revelation (“Record does not exist”, for example). These messages are converted to their Advanced Revelation counterparts for display.

SQL Server errors that do not have exact parallels in Advanced Revelation display either an Advanced Revelation-style message or the text and codes from an SQL Server message. In either case, the SQL Server error message number displays at the bottom of the message window so you can trace the error on the SQL Server, if necessary.

Error Checking within Programs

When you use R/BASIC programs and embedded SQL, you can test program execution status and determine how errors should be handled by the program. To check program execution status:

- In Advanced Revelation embedded SQL, codes for errors that occur during processing are returned in the `SQLSTATE` variable. You can find the full range of `SQLSTATE` values in a file called `INCLUDE`. Any record of the name `SQLSTATES_nnn` (where `nnn` is a number) contains a list of `SQLSTATE` error explanations.
- In R/BASIC programs, codes for errors that occur during processing are returned in `@FILE.ERROR`. An explanation of filing system error codes can be found in the `INCLUDE` record named `FSERRORS_200`.

For more information, see the topic “Testing Execution Status (the `SQLSTATE` Variable)” in the chapter “Programming with SQL” in *SQL*. Also see the chapter “Programming with R/BASIC” in *R/BASIC*.

Tracing Errors

When an Advanced Revelation process causes an error on the SQL Server, information about the error is logged in a file called `MSSQL_LOG`. `MSSQL_LOG` is located on the `REVBOOT` volume.

To view the information in the `MSSQL_LOG`, enter the TCL command:

```
LIST MSSQL_LOG REPORT
```

The resulting report displays the contents of various fields in the `MSSQL_LOG`. To determine what error has occurred, check the `ERR_NO` column:

- If the number in the `ERR_NO` column is under 10000, the error is defined in the `SYSMESSAGES` file on the SQL Server.
- If the number in the `ERR_NO` column is 10000 or greater, refer to the error information in the Programmer's Reference documentation provided with the SQL Server. SQL Server error values are defined in the `DB-LIBRARY` header file `SQLFRONT.H`.

Look at the **SCRIPT** column in the **MSSQL_Log** report to see the Transact-SQL statement that caused the error.

- ☛ If the server trace feature is on for any SQL Server volumes, the **MSSQL_LOG** file may contain records of the Transact-SQL queries that Advanced Revelation processes have run without error. If there is no number in the **ERR_NO** column, the Transact-SQL query was successful.

Using the MSSQL_LOG File

The **MSSQL_LOG** is located on the **REVBOOT** volume. The **MSSQL_LOG** contains records of Transact-SQL commands that Advanced Revelation processes have sent to the SQL Server. It is updated under the following circumstances:

- Each time an Advanced Revelation process causes an error on the SQL Server, a record with information about the error is logged.
 - ☛ See “Error Handling” above for more information.
- If Advanced Revelation SQL Server tracing is on for a volume, then, each time an Advanced Revelation process accesses a SQL Server table in that volume the resulting Transact-SQL commands are logged.
 - ☛ You can turn on the Advanced Revelation server trace function for a SQL database when you set up the SQL Server volume in Advanced Revelation. See “Setting Up Advanced Revelation Volumes for the SQL Server” for more information.

You can use the information in the **MSSQL_LOG** file to see how Advanced Revelation processes are communicating with the SQL Server and to review any Transact-SQL command that caused an error to occur on the SQL Server.

To view the information in the **MSSQL_LOG**, enter the TCL command:

```
LIST MSSQL_LOG REPORT
```

The resulting report displays the contents of various fields in the **MSSQL_LOG**. Look at the **SCRIPT** column to see the Transact-SQL statement that Advanced Revelation is sending to the SQL Server.

MSSQL_LOG File Maintenance

Over time the **MSSQL_LOG** file may grow to a considerable size. Periodically you should clear the **MSSQL_LOG** file to reduce its size and free up space on your hard disk. This is not handled automatically, so the system administrator may want to implement a clearing procedure that is run on a regular basis.

Freeing Memory

The TSR program DBNMPPIPE.EXE occupies approximately 75K of RAM. If your network software requires that you load DBNMPPIPE before you can access the SQL Server, there is a way to free this memory when you finish using the SQL Server. Just enter the following command at the operating system prompt:

```
ENDDBLIB
```

MSSQL_C Routine

The MSSQL_C routine enables you to access the SQL Server directly from an R/BASIC program. All calls to MSSQL_C must pass:

- A function code as the first argument.
- The data segment as the last argument. For information about establishing the data segment, see the MSSQL.INIT\$ function.

MSSQL_C returns errors in ERRFLAG and ERRSTR. If ERRFLAG = ERR_NONE, no errors occurred. ERRFLAG = ERR_NO_MORE_ROWS indicates that there are no more rows to be returned and is the only normal non-zero error. ERRSTR contains any messages sent from the server to the client. Normally this is only when ERRFLAG is set, but not always. ERRFLAG must be the third to the last argument and ERRSTR must be the second to last argument.

In this section, DB-LIBRARY functions are represented in lowercase letters and macros are represented in uppercase.

The 24 functions that follow are all EQUATES in the MSSQL_EQUATES record located in the INCLUDE file.

Function 1 - MSSQL.INIT\$

MSSQL.INIT\$ must be the first function called against a fresh data segment. The data segment is located in the VERBS file with the record name MSSQL_C_DAT. This record can be read into an R/BASIC variable using an R/BASIC READ and is then initialized with this function. This function also verifies that the TSR is loaded and installs the error and message handlers.

```
MSSQL_C (MSSQL.INIT$, ERRFLAG, ERRSTR, DATA)
```


Parameter	Direction	Description
MSSQL.INIT\$	Passed	Function code (= 1)
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Sample code for a shell that uses MSSQL.INIT\$ to initialize a data segment is located at the end of this section.

Function 2 - MSSQL.OPEN\$

MSSQL.OPEN\$ establishes a standard connection to a server. The APPLICATION and HOST parameters appear in the Sysprocesses table and in the sp_who listing.

```
MSSQL_C (MSSQL.OPEN$, SERVER, USER, PASSWORD, APPLICATION, HOST, ⇐
        DBPROCESS, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.OPEN\$	Passed	Function code (= 2)
SERVER	Passed	Name of the server to connect to
USER	Passed	Name to use in logging into the server
PASSWORD	Passed	Password of the user
APPLICATION	Passed	Name of this application
HOST	Passed	Name of this host
DBPROCESS	Returned	The established connection handle
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 3 - MSSQL.EXEC.LATENT\$

Execute an SQL script. The resulting rows will then be read in one row at a time using MSSQL.NEXT.ROW\$. This is usually used when:

- An internally generated SQL select statement needs to be processed.
- The data being returned is of a known type.
- The number of rows to be returned is unknown.

```
MSSQL_C (MSSQL.EXEC.LATENT$, DBPROCESS, SQL.SCRIPT, ERRFLAG, ⇐  
ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXEC.LATENT\$	Passed	Function code (= 3)
DBPROCESS	Passed	The SQL server connection
SQL.SCRIPT	Passed	The SQL script to execute
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Sample code for a shell that uses MSSQL.EXEC.LATENT\$ is located at the end of this section.

Function 4 - MSSQL.EXEC.NONE\$

Execute an SQL script. All the results will be immediately processed, and if any rows are returned from the server, an error is generated and the script is cancelled. MSSQL.EXEC.NONE\$ is used when an internally generated SQL statement is sent that should not return any rows.

```
MSSQL_C (MSSQL.EXEC.NONE$, DBPROCESS, SQL.SCRIPT, ERRFLAG, ⇐  
ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXEC.NONE\$	Passed	Function code (= 4)
DBPROCESS	Passed	The SQL server connection
SQL.SCRIPT	Passed	The SQL script to execute
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 5 - MSSQL.EXEC.READ\$

Execute an SQL script and read in all returned rows. The returned rows will be separated by @RM's. Columns are separated by @FM's. If no rows were returned, ERRFLAG is set to ERR_NO_MORE_ROWS. MSSQL.EXEC.READ\$ is usually used when:

- An internally generated SQL select statement needs to be processed.
- The data being returned is of a known type.
- The number of rows to be returned is known.

```
MSSQL_C (MSSQL.EXEC.READ$, DBPROCESS, SQL.SCRIPT, RECORD, ⇐
ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXEC.READ\$	Passed	Function code (= 5)
DBPROCESS	Passed	The SQL server connection
SQL.SCRIPT	Passed	The SQL script to execute
RECORD	Returned	The returned rows
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 6 - MSSQL.EXEC.READTEXT\$

Execute an SQL script and read in all returned rows into a single field of a record. This is used to send an SQL script of READTEXT statements and then read in the returned Text/Image information into an already existing record. OFFSET is the position in RECORD to start reading the data. RECORD must be pre-sized to hold all of the returned data.

```
MSSQL C (MSSQL.EXEC.READTEXT$, DBPROCESS, SQL.SCRIPT, OFFSET, ⇨  
RECORD ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXEC.READTEXT\$	Passed	Function code (= 6)
DBPROCESS	Passed	The SQL server connection
SQL.SCRIPT	Passed	The READTEXT SQL script to execute
OFFSET	Passed	Where in RECORD to start reading the data into
RECORD	Passed	A pre-sized record to hold the data returned
	Returned	The returned data
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 7 - MSSQL.EXEC.SCRIPT\$

Execute an SQL script. The results will be processed via MSSQL.RESULTS\$ and MSSQL.RESULT.ROW\$ functions. Use MSSQL.EXEC.SCRIPT\$ to process random SQL statements.

```
MSSQL C (MSSQL.EXEC.SCRIPT$, DBPROCESS, SQL.SCRIPT, ERRFLAG, ⇨  
ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXEC.SCRIPT\$	Passed	Function code (= 7)
DBPROCESS	Passed	The SQL server connection
SQL.SCRIPT	Passed	The SQL script to execute
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Sample code for a shell that uses MSSQL.EXEC.SCRIPT\$ is located at the end of this section.

Function 8 - MSSQL.NEXT.ROW\$

Read in the next row from the latent SQL script in progress. Use MSSQL.NEXT.ROW\$ in conjunction with the MSSQL.EXEC.LATENT\$ function. ERRFLAG is set to ERR_NO_MORE_ROWS when there are no more rows in the current batch.

MSSQL_C (MSSQL.NEXT.ROW\$, DBPROCESS, RECORD, ERRFLAG, ERRSTR, DATA)
--

Parameter	Direction	Description
MSSQL.NEXT.ROW\$	Passed	Function code (= 8)
DBPROCESS	Passed	The SQL server connection
RECORD	Returned	The next row returned from the server
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 9 - MSSQL.RESULTS\$

Return the results of the next SQL command in progress. Use MSSQL.RESULTS\$ in conjunction with MSSQL.EXEC.SCRIPT\$ and MSSQL.RESULT.ROW\$.
ERRFLAG is set to ERR_NO_MORE_ROWS when there are no more commands in the current batch.

MSSQL_C (MSSQL.RESULTS\$, DBPROCESS, RESULTS, ERRFLAG, ERRSTR, DATA)

Parameter	Direction	Description
MSSQL.RESULTS\$	Passed	Function code (= 9)
DBPROCESS	Passed	The SQL server connection
RESULTS	Returned	The results of the next command. See below
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

The RESULTS variable has the following structure:

Field	Definition
1 DBCMDROW	If true, this statement can return rows
2 DBROWS	If true, this statement did return rows
3 DBCOUNT	Number of rows affected by this statement
4 dbnumcols	Number of columns in each regular row returned
5 dbcollen	Length of data in the columns
6 dbcolname	Name of the columns
7 dbcoltype	Type of data in the columns
8 DBNUMORDERS	Number of ORDER BY clauses
9 dbordercol	Column number of the ORDER BY clause
10 dbnumcompute	Number of COMPUTE rows
11 dbnumalts	Number of columns in the COMPUTE rows
12 dbaltlen	Length of data in the COMPUTE columns
13 dbalttype	Type of data in the COMPUTE columns

(Continued)

Field	Definition
14 dbaltop	Type of operation in the COMPUTE columns
15 dbaltcolid	Column number of the COMPUTE columns
16 dbbylist	BY clause assoc. with the COMPUTE rows
17 dbbylist	Column number of the COMPUTE row BY clause

Fields 5, 6, and 7 are associated multivalues with each value corresponding to one regular row column.

Field 9 is multivalued with each value corresponding to one ORDER BY clause.

Fields 11 through 17 are associated multivalues with each value corresponding to one COMPUTE row.

Fields 12 through 15 are associated multi-subvalues with each subvalue corresponding to one column of the COMPUTE row.

Field 17 is multi-subvalued with each subvalue corresponding to one BY clause of the COMPUTE row.

Function 10 - MSSQL.RESULT.ROW\$

Read in the next row from the SQL script in progress. This is used in conjunction with the MSSQL.EXEC.SCRIPT\$ and MSSQL.RESULTS\$ functions. ERRFLAG is set to ERR_NO_MORE_ROWS when there are no more rows in the current command.

```
MSSQL C (MSSQL.RESULT.ROW$, DBPROCESS, ROW_TYPE, RECORD, ERRFLAG, ⇨
ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.RESULT.ROW\$	Passed	Function code (= 10)
DBPROCESS	Passed	The SQL server connection
ROW_TYPE	Returned	Value greater than zero identifies a COMPUTE row result
RECORD	Returned	The next row returned from the server
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

A value greater than zero in the ROW_TYPE variable indicates COMPUTE row results. Because a query can have multiple COMPUTEs, each COMPUTE row is assigned a different ROW_TYPE value. The ROW_TYPE value is used as a reference value to locate information in fields 10 through 17 in the MSSQL.RESULTS\$ RESULTS variable.

Function 11 - MSSQL.WRITETEXT\$

Write out Text/Image data. This is used as a more efficient means of writing large Text/Image data fields to a server. The TEXTPTR value must already have been obtained from the server.

```
MSSQL_C (MSSQL.WRITETEXT$, DBPROCESS, OBJECT.NAME, TEXTPTR, ⇨  
        OFFSET, LENGTH, RECORD, ERRFLAG, ERRSTR, DATA)
```

- OBJECT.NAME is the name of the column that this Text data is associated with.
- TEXTPTR is the DB-LIBRARY text pointer to the Text data.
- OFFSET is the offset into RECORD to the start of the data to be written out.
- LENGTH is the length of the data in RECORD to be written out.
- RECORD contains the data to be written out.

Parameter	Direction	Description
MSSQL.WRITETEXT\$	Passed	Function code (= 11)
DBPROCESS	Passed	The SQL server connection
OBJECT.NAME	Passed	The name of the column
TEXTPTR	Passed	The DB-LIBRARY text pointer to the Text data
OFFSET	Passed	The offset into RECORD to the start of the data
LENGTH	Passed	The length of the data in RECORD
RECORD	Passed	The record with the data to send
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 12 - MSSQL.CANQUERY\$

Cancel any rows pending from the most recent executed query.

```
MSSQL_C (MSSQL.CANQUERY$, DBPROCESS, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.CANQUERY\$	Passed	Function code (= 12)
DBPROCESS	Passed	The SQL server connection
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 13 - MSSQL.CANCEL\$

Cancel the current command batch.

```
MSSQL_C (MSSQL.CANCEL$, DBPROCESS, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.CANCEL\$	Passed	Function code (= 13)
DBPROCESS	Passed	The SQL server connection
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 14 - MSSQL.CLOSE\$

Close down a server connection.

```
MSSQL_C ( MSSQL.CLOSE$, DBPROCESS, ERRFLAG, ERRSTR, DATA )
```

Parameter	Direction	Description
MSSQL.CANCEL\$	Passed	Function code (= 14)
DBPROCESS	Passed	The SQL server connection
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 15 - MSSQL.OPEN.COMMIT\$

Establish a two-phase commit connection to a server.

```
MSSQL_C (MSSQL.OPEN.COMMIT$, SERVER, USER, PASSWORD, HOST, ⇨  
DBPROCESS, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.OPEN.COMMIT\$	Passed	Function code (= 15)
SERVER	Passed	Name of the server to connect to
USER	Passed	Name to use in logging into the server
PASSWORD	Passed	Password of the user
HOST	Passed	Name of this host
DBPROCESS	Returned	The established connection handle
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 16 - MSSQL.START.XACT\$

MSSQL.START.XACT begins a two-phase commit transaction.

```
MSSQL_C (MSSQL.START.XACT$, DBPROCESS, APPLICATION, XACT.NAME, ⇨
SITE.COUNT, COMMID, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.START.XACT\$	Passed	Function code (= 16)
DBPROCESS	Passed	The SQL server two-phase commit connection
APPLICATION	Passed	Application name
XACT.NAME	Passed	Transaction name
SITE.COUNT	Passed	Count of participating sites
COMMID	Returned	The two-phase commit id
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 17 - MSSQL.BUILD.XACT.STRING\$

MSSQL.BUILD.XACT.STRING is used by MSSQL.START.XACT in building a transaction string

```
MSSQL_C (MSSQL.BUILD.XACT.STRING$, DBPROCESS, XACT.NAME, ⇨
SERVICE.NAME, COMMID, RESULT, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.BUILD.XACT.STRING\$	Passed	Function code (= 17)
DBPROCESS	Passed	The SQL server two-phase commit connection
XACT.NAME	Passed	Transaction name
SERVICE.NAME	Passed	Service name
COMMID	Passed	The two-phased commit id
RESULT	Returned	The built transaction string
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 18 - MSSQL.COMMIT.XACT\$

MSSQL.COMMIT.XACT\$ signals a commit for the two-phase commit.

MSSQL_C (MSSQL.COMMIT.XACT\$, DBPROCESS, COMMID, ERRFLAG, ERRSTR, ⇨ DATA)

Parameter	Direction	Description
MSSQL.COMMIT.XACT\$	Passed	Function code (= 18)
DBPROCESS	Passed	The SQL server two-phase commit connection
COMMID	Passed	The two-phased commit id
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 19 - MSSQL.REMOVE.XACT\$

MSSQL.REMOVE.XACT removes an entry from the two-phase commit table.

```
MSSQL_C (MSSQL.REMOVE.XACT$, DBPROCESS, XACT.SITES, ERRFLAG, ⇐
ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.REMOVE.XACT\$	Passed	Function code (= 19)
DBPROCESS	Passed	The SQL server two-phase commit connection
COMMITID	Passed	The two-phased commit id
XACT.SITES	Passed	Transaction sites
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 20 - MSSQL_ABORT_XACT\$

MSSQL_ABORT_XACT\$ signals a rollback for the two-phase commit.

```
MSSQL_C (MSSQL.ABORT.XACT$, DBPROCESS, COMMITID, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.ABORT.XACT\$	Passed	Function code (= 20)
DBPROCESS	Passed	The SQL server two-phase commit connection
COMMITID	Passed	The two-phased commit id
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 21 - MSSQL.STAT.XACT\$

**MSSQL_C (MSSQL.STAT.XACT\$, DBPROCESS, COMMID, STAT, ERRFLAG, ⇨
ERRSTR, DATA)**

Parameter	Direction	Description
MSSQL.STAT.XACT\$	Passed	Function code (= 21)
DBPROCESS	Passed	The SQL server two-phase commit connection
COMMID	Passed	The two-phased commit id
STAT	Returned	Transaction status
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 22 - MSSQL.CLOSE.COMMIT\$

MSSQL.CLOSE.COMMIT\$ closes a two-phase commit connection.

MSSQL_C (MSSQL.CLOSE.COMMIT\$, DBPROCESS, ERRFLAG, ERRSTR, DATA)

Parameter	Direction	Description
MSSQL.CLOSE.COMMIT\$	Passed	Function code (= 22)
DBPROCESS	Passed	The SQL server two-phase commit connection
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 23 - MSSQL.EXIT\$

Close down all open server connections and terminate use of this data segment.

```
MSSQL_C (MSSQL.EXIT$, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.EXIT\$	Passed	Function code (= 23)
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

Function 24 - MSSQL.SETTIME\$

Set the number of seconds to wait for a response from the server.

```
MSSQL_C (MSSQL.SETTIME$, TIME, ERRFLAG, ERRSTR, DATA)
```

Parameter	Direction	Description
MSSQL.SETTIME\$	Passed	Function code (= 24)
TIME	Passed	The number of second to wait for
	Returned	The previous number of seconds waited
ERRFLAG	Returned	Return status
ERRSTR	Returned	Errors/messages from the server
DATA	Passed	The data segment

About the ERRFLAG Parameter

The ERRFLAG parameter can have the following values. Also, if the installed error handler is called, ERRFLAG will contain the error number passed to it. ERRFLAG will contain the highest error number that occurred during the call. All error numbers passed to the error handler are greater than the error numbers below. This allows the error handler errors to take priority over the C generated errors.

ERRFLAG will not be set if just the message handler gets called. This can happen if an SQL script does a PRINT statement. This can happen during the execution of an SQL script, a stored procedure, a trigger, or a rule. If the intent of the PRINT statement was to indicate an error condition, then the ERRFLAG will not be set. Instead, a RAISEERROR statement should have been performed, which calls the error handler instead and allows an error number to be passed through the ERRFLAG. This number is then used to generate an Advanced Revelation message id.

ERR_NONE (0)	<i>No errors returned.</i> No errors encountered during last MSSQL_C call.
ERR_NO_MORE_ROWS (1)	<i>No more rows during request to read row.</i> No more rows are available from the Server. This is the only normal error status return.
ERR_ROWS_RET (2)	<i>Rows returned when none expected.</i> Rows were returned during the MSSQL.EXEC.NONE\$ call. The SQL statement executed returned rows. Either use MSSQL.EXEC.READ\$ or change the SQL statement to not return rows.
ERR_CODE (3)	<i>Invalid function code in first parameter.</i> The first parameter was not between 1 and 24. Pass a valid function number.
ERR_NO_ARGS (4)	<i>Invalid number of arguments.</i> Each function requires a specific number of arguments. The wrong number of arguments were passed for the function number.
ERR_DBINIT (5)	<i>dbinit failure.</i> The call to dbinit failed. Currently the only way dbinit can fail is if the DB-LIBRARY TSR has not been installed. Exit Advanced Revelation and install the TSR.
ERR_DBSETTIME (6)	<i>dbsettime failure.</i> The call to dbsettime failed.
ERR_DBLOGIN (7)	<i>dblogin failure.</i> The call to dblogin failed. Usually, when dblogin fails the error handler is also called, so ERRFLAG will contain the error number passed to the error handler. This is probably an out-of-memory condition.
ERR_DBSETLUSER (8)	<i>DBSETLUSER failure.</i> The call to DBSETLUSER failed.
ERR_DBSETLPWD (9)	<i>DBSETLPWD failure.</i> The call to DBSETLPWD failed.
ERR_DBSETLAPP (10)	<i>DBSETLAPP failure.</i> The call to DBSETLAPP failed.

- ERR_DBSETLHOST**
(11) *DBSETLHOST failure.* The call to **DBSETLHOST** failed.
- ERR_DBOPEN**
(12) *dbopen failure.* The call to **dbopen** failed. Usually, when **dbopen** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler. **dbopen** can fail for a variety of reasons, including: out of memory, invalid server name, invalid user name or password, etc.
- ERR_DBPROCESS**
(13) *Invalid passed DBPROCESS.* The passed **DBPROCESS** is invalid. The current version of **DB-LIBRARY** allocates memory and returns a pointer to that memory as the **DBPROCESS**. A check is made to make sure the passed **DBPROCESS** resides in the allocated memory space.
- ERR_DBCMD**
(14) *dbcmd failure.* The call to **dbcmd** failed. Usually, when **dbcmd** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler. **dbcmd** can fail for a variety of reasons, including: out of memory, invalid SQL script, privilege violation, etc.
- ERR_DBSQLEXEC**
(15) *dbsqlxec failure.* The call to **dbsqlxec** failed. Usually, when **dbsqlxec** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler. **dbsqlxec** can fail for a variety of reasons, including: out of memory, invalid SQL script, privilege violation, etc.
- ERR_DBSQLOK**
(16) *dbsqllok failure.* The call to **dbsqllok** failed. Usually, when **dbsqllok** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler. This can fail for a variety of reasons, including: out of memory, invalid SQL script, privilege violation, etc.
- ERR_DBRESULTS**
(17) *dbresults failure.* The call to **dbresults** failed. Usually, when **dbresults** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler.
- ERR_DBNEXTROW**
(18) *dbnextrow failure.* The call to **dbnextrow** failed. Usually, when **dbnextrow** fails the error handler is also called, so **ERRFLAG** will be the error number passed to the error handler.
- ERR_DBTXPLEN**
(19) *Text pointer data not DBTXPLEN bytes long.* The passed Text Pointer is not the correct size. Text Pointers are a specific number of binary bytes long.

ERR_DBWRITETEXT (20)	<i>dbwritetext failure.</i> The call to dbwritetext failed. Usually, when dbwritetext fails the error handler is also called, so ERRFLAG will be the error number passed to the error handler.
ERR_DBMORETEXT (21)	<i>dbmoretext failure.</i> The call to dbmoretext failed. Usually, when dbmoretext fails the error handler is also called, so ERRFLAG will be the error number passed to the error handler.
ERR_READTEXT_SIZE (22)	<i>ReadText data larger than passed record.</i> The size of the record to store the ReadText data is smaller than the data received from the server. This can only occur during a MSSQL.EXEC.READTEXT\$ call. The record should have been pre-sized to the size of the text sent back by the readtext statements being executed.
ERR_DBCANQUERY (23)	<i>dbcانquery failure.</i> The call to dbcانquery failed.
ERR_DBCANCEL (24)	<i>dbcancel failure.</i> The call to dbcانquery failed.
ERR_OPEN_COMMIT (25)	<i>open_commit failure.</i> The call to open_commit failed.
ERR_START_XACT (26)	<i>start_xact failure.</i> The call to start_xact failed.
ERR_COMMIT_XACT (27)	<i>commit_xact failure.</i> The call to commit_xact failed.
ERR_REMOVE_XACT (28)	<i>remove_xact failure.</i> The call to remove_xact failed.
ERR_ABORT_XACT (29)	<i>abort_xact failure.</i> The call to abort_xact failed.
ERR_STAT_XACT (30)	<i>stat_xact failure.</i> The call to stat_xact failed.

About the ERRSTR Parameter

The ERRSTR parameter returns all information passed to the error and message handlers. Each time either of the handlers is called, a field is added to ERRSTR. The format of each field added is:

Value	Message Handler	Error Handler
1	Literal "msg"	Literal "err"
2	DBPROCESS message occurred under	DBPROCESS error occurred under
3	Severity of the message	Severity of the error
4	Message number	Error number
5	Message state	OS error number
6	Message text	Error text
7	Not used	OS error text

Examples of Function Usage

The following is a shell for using MSSQL.INIT\$ to initialize a data segment and open a DBPROCESS:

```

OPEN "VERBS" TO VERBS FH ELSE
  MSG("Cannot open file VERBS.", '', '', '')
  RETURN
END

READ DATA SEGMENT FROM VERBS FH, "MSSQL_C_DAT" ELSE
  MSG("Record MSSQL_C_DAT doesn't exist in file VERBS.", '', '', '')
  RETURN
END

/* Initialize the fresh data segment */
MSSQL_C(MSSQL.INIT$, ERRFLAG, ERRSTR, DATA_SEGMENT)
IF ERRFLAG THEN
  GOSUB ERROR_HANDLER
  RETURN
END

/* Establish a connection to the server, this process will assign DBPROCESS
a process number if successful */
MSSQL_C(MSSQL.OPEN$, "SQLSERVER", "sa", '', "USER APP", '', DBPROCESS, ER-
RFLAG, ERRSTR, DATA_SEGMENT)
IF ERRFLAG THEN
  GOSUB ERROR_HANDLER
  RETURN
END

```

The following is a shell for closing a manually opened DBPROCESS:

```
/* Close the DBPROCESS on the server */
MSSQL_C(MSSQL.CLOSE$, DBPROCESS, ERRFLAG, ERRSTR, DATA_SEGMENT)
IF ERRFLAG THEN
    GOSUB ERROR_HANDLER
END
/* Make sure the DATA_SEGMENT is clear out of memory */
DATA_SEGMENT = ''
```

The following is a shell for using MSSQL.EXEC.LATENT\$:

```
/* SQL.SCRIPT contains the generated SQL script to be executed. */
MSSQL_C( MSSQL.EXEC.LATENT$, DBPROCESS, SQL.SCRIPT, ERRFLAG, ERRSTR, DATA )
IF ERRFLAG THEN
    GOSUB MSSQL.ERROR
    RETURN
END
LOOP
    MSSQL_C( MSSQL.NEXT.ROW$, DBPROCESS, RECORD, ERRFLAG, ERRSTR, DATA)
UNTIL ERRFLAG = ERR_NO_MORE_ROWS$
    IF ERRFLAG THEN
        GOSUB MSSQL.ERROR
        RETURN
    END
/* Process the current RECORD. */
REPEAT
```

The following is a shell for using MSSQL.EXEC.SCRIPT\$:

```
/* SQL.SCRIPT contains the generated SQL script to be processed. */
MSSQL_C( MSSQL.EXEC.SCRIPT$, DBPROCESS, SQL.SCRIPT, ERRFLAG,
    ERRSTR, DATA )
IF ERRFLAG THEN
    GOSUB MSSQL.ERROR
    RETURN
END
LOOP
    MSSQL_C (MSSQL.RESULTS$, DBPROCESS, RESULTS, ERRFLAG, ERRSTR,
        DATA)
UNTIL ERRFLAG = ERR_NO_MORE_ROWS$
    IF ERRFLAG THEN
        GOSUB MSSQL.ERROR
        RETURN
    END
/* Process the current RESULTS set. */
LOOP
    MSSQL_C( MSSQL.RESULT.ROW$, DBPROCESS, ROW_TYPE, RECORD, ERRFLAG,
        ERRSTR, DATA )
UNTIL ERRFLAG = ERR_NO_MORE_ROWS$
    IF ERRFLAG THEN
        GOSUB MSSQL.ERROR
        RETURN
    END
/* Process the current RECORD of the current RESULT set. */
REPEAT
REPEAT
```

DB-LIBRARY Functions and Macros Used

The Advanced Revelation MSSQL_C routine currently uses the following DB-LIBRARY functions and MACROS:

abort_xact	dbclose	dbinit	DBROWS
build_xact_string	dbcmd	dblogin	DBSETLAPP
close_commit	DBCMDROW	dbmoretext	DBSETLHOST
commit_xact	dbcollen	dbmsghandle	DBSETLPWD
dbadata	dbcollname	dbnextrow	DBSETLUSER
dbadlen	dbcotype	dbnumalts	dbsettime
dbaltcolid	DBCOUNT	dbnumcols	dbsqlxec
dbaltlen	dbdata	dbnumcompute	dbsqllok
dbaltop	dbdatlen	DBNUMORDERS	dbwritetext
dbalttype	dberrhandle	dbopen	open_commit
dbbylist	dbexit	dbordercol	remove_xact
dbcancel	dbfreebuf	dbprtype	start_xact
dbcquery	dbfreelogin	dbresults	stat_xact

NETSERVERENUM Routine

The Advanced Revelation routine NETSERVERENUM locates SQL Servers on the network by calling the LAN Manager Applications Program Interface function NetServerEnum.

✚ NETSERVERENUM works on both LAN Manager and Novell networks.

```
NETSERVERENUM(error, servers)
```

NETSERVERENUM returns values for both *error* and *servers*. If *error* is:

- 0, no errors were encountered. *Servers* contains one record for each server on the network.
- 1, the version of DOS being run is less than 3.0.
- Any other positive value, that value is the error number returned from NetServerEnum.
- A negative value, the absolute value of *error* is the number of servers that Advanced Revelation was not able to buffer up. *Servers* contains one record for each server on the network.

The format of each record is:

Field	Description
1	Server name
2	Server version major
3	Server version minor
4	Server type
5	Server comment

Chapter N 1

Introduction to R/BASIC

As you begin to develop more sophisticated applications, you will find yourself taking advantage of Advanced Revelation's powerful programming features. In this and subsequent chapters you will find a complete description of R/BASIC, the language used to create symbolic dictionary records and stand-alone programs.

This chapter introduces you to the general concepts of using R/BASIC in your applications. Later chapters in this section provide you with detailed information on programming in Advanced Revelation.

R/BASIC Overview N 1 . 1



R/BASIC Overview

R/BASIC is Advanced Revelation's programming language. Although the name implies that R/BASIC is Advanced Revelation's variant of the popular BASIC programming language, R/BASIC actually incorporates features and functions found in a variety of languages. In addition, R/BASIC is flexible enough to allow you to write programs in the styles of other languages.

R/BASIC also includes a number of unique features. Some of these, such as the dynamic array operators and conversion functions, are keyed to the Advanced Revelation environment. Other features found in R/BASIC are conveniences which would be welcome in any language: lack of data typing, lock and unlock statements, a variety of string handling tools, and constructs for structured programming, to name a few.

Why Use R/BASIC?

Advanced Revelation already provides very powerful tools in the form of retrieval languages and an applications generator. Given these, why would you need to resort to R/BASIC?

The answer lies in recognizing the function of Advanced Revelation's tool kit. An application processor such as Paint and Window provide the means for building what in essence is a structure for the input of data into files. This function is easily seen in the creation of entry screens and menus.

On the other hand, the function of a retrieval language such as SQL or R/LIST is to take data from a file and put it onto the screen or paper in an organized, cohesive form. While these retrieval languages are powerful enough to permit great variation in the organization of this information, their basic function remains essentially that of a report writer.

You can put together complete applications using only these tools. An example might be an index of documents or books. You create some entry screens and menus, and the data is input in a standardized format. Users then selectively retrieve information about this index by using the features of SQL, R/LIST, Merge, and other reporting tools.

Many applications, however, require not just the input and output of data on file, but also the manipulation of that data. A familiar example could be an invoicing system in which users enter information about orders, and then the computer calculates invoice amounts and totals.

To create such an application, you must use R/BASIC. In this instance, an R/BASIC program could be developed which would calculate and sum the different prices on an invoice, calculate tax if applicable, and print subtotals and totals on the invoice itself. R/BASIC could also be used in the posting of invoices, the tracking of payments, and other tasks such as those requiring mathematical, comparison, or filing functions.

The range of R/BASIC, however, is much greater than what is implied by this simple example. R/BASIC is used throughout Advanced Revelation. It appears in symbolic dictionary items, where it is used to link together files, format output, calculate quantities, and more. R/BASIC is also the basis for virtually all the commands which are used every day in Advanced Revelation — RECORDCOPY, SUM, SELECT, and EDIT. In fact, Advanced Revelation itself might be viewed as an engine to run R/BASIC programs.

How R/BASIC Programs are Created and Executed

R/BASIC programming comes in two forms: symbolic dictionary items and stand-alone programs.

Symbolic dictionary items are the first example of R/BASIC. They differ from standard dictionary items in that they do not directly refer to data on file, but rather use a formula as their definition. This formula is actually an R/BASIC program. It can be very short (one line), or very complex (up to 32K of source code). Programs created as symbolic dictionary items are executed when those dictionary items are called by some other process such as R/LIST, the application windows, or another R/BASIC program.

R/BASIC programs can also be created as independent units and called by other programs or by code and command processes. Stand-alone programs are kept as records in an Advanced Revelation file. No distinction is made between data and program files, but it is not a good practice to keep R/BASIC programs in data files or in the VOC file. Many programmers use a unique naming convention for files containing R/BASIC programs. For example, they might create a file named BP.

The process of actually creating a program requires several steps:

- Use the option **Tools-Developer-Edit** or the TCL command EDIT to call Advanced Revelation's editor. Using the editor, you write a series of program instructions. This set of instructions is the source code.
- Call the compiler, which translates the source code into a format interpretable by Advanced Revelation. This is called the object code. For stand-alone programs, the object code is stored in the same file as the source code, but it has a dollar sign (\$) in front of the name.

At this point, the TCL command RUN will execute the stand-alone program and return to TCL when finished. You also have the option of putting an entry in the VOC file for this program by using the MAKEVOC command. Users can then simply type in the program name at TCL to execute it.

For example, these steps (executed from TCL) illustrate a typical scenario:

```
EDIT BP INVOICE.PGM
```

Create or edit the source code for a program called INVOICE.PGM stored in the BP file.

```
COMPILE BP INVOICE.PGM
```

Translate the source code into Advanced Revelation object code. The result is an item in the BP file called \$INVOICE.PGM.

```
RUN BP INVOICE.PGM
```

Execute the program.

```
MAKEVOC BP INVOICE.PGM
```

Put an entry for this program in the VOC file. From this point, the program can be run by typing INVOICE.PGM at TCL.

- ✎ The RUN and MAKEVOC commands pay attention only to the object code (the \$ form of the program), so that the source code can be removed from the file. If the program needs to be changed, however, the source code needs to be edited and then recompiled.

Symbolic dictionary items follow roughly the same sequence of events. The source code is edited using the Dictionary window. This code is compiled automatically when the dictionary item is filed. The resulting object code is not stored separately, but is kept in a field in the same dictionary item.

Listing R/BASIC Programs

A command is provided in Advanced Revelation to create special formatted listings of R/BASIC programs. This command, BLIST, will send to the printer a version of the source code in which all loops and multi-lined constructions are properly indented. This can help in both developing and documenting an R/BASIC program.

What You Should Know When Using R/BASIC

Before beginning to work with R/BASIC, you should review the fundamental concepts underlying R/BASIC programs. These concepts are discussed in the chapter "Programming with R/BASIC."

You should also be comfortable with ways in which Advanced Revelation stores data on file. The dynamic array structure of records in an Advanced Revelation file is important, as are the various internal formats by which dates, times, and decimal numbers are stored. Dynamic arrays are discussed in "Programming with R/BASIC"; internal data formats are discussed under ICONV and OCONV in the "R/BASIC Command Reference" chapter.

Both symbolic fields and stand-alone programs can access data by way of dictionary field names. Understanding how R/BASIC does this is very helpful. This topic is covered under "Interfacing Dictionaries and R/BASIC" in the chapter "Programming with R/BASIC." The CALCULATE and {} (braces) functions accomplish this task, so a review of these in the chapter "R/BASIC Command Reference" will help in writing programs that take advantage of this powerful R/BASIC capability.

Other concepts are useful in R/BASIC as well. One example is the concept of recursive execution whereby a program can call itself. A program may also call a TCL process by preceding the command with PERFORM or EXECUTE. The @-variables also play an important role in programming. For more information, see "Special R/BASIC Variables" in "Programming with R/BASIC."

If an application is being developed for use under a local area network, you should review the section *Networking*, as well as LOCK and UNLOCK in the "R/BASIC Command Reference" chapter.

It is not necessary to master all of these concepts in order to develop applications using R/BASIC. However, as you need for sophisticated programming increases, you will find it useful to investigate the full range of tools available in the language.

Chapter N 2

Features of R/BASIC

In this chapter you will find a list of all R/BASIC commands and functions, organized by category. Next to each entry is a short description of the purpose of the statement or function.

This chapter is useful if you know generally what you want to accomplish — for example, you know that you need to read a record from a file — but are unsure of what R/BASIC commands to use. You can also treat this chapter as a learning aid. Browse through the list of commands to discover the capabilities of R/BASIC, and to see how the commands fit together.

If you need more conceptual information about programming in R/BASIC, see the chapter "Programming with R/BASIC."

Features of R/BASIC N2.1



Features of R/BASIC

R/BASIC features consists of the statements, functions, and operators that make up an R/BASIC program. Features are listed in categories by their topic. A brief description follows each item.

Compiler

Compiler commands direct the compiler to perform certain tasks while a program is being compiled into object code. Compiler commands are not used in the execution of the program.

;	Delimits a new line (allows multiple commands to appear on the same line).
\$INSERT	Inserts source code from another program into the current program at this point.
, !, REM, /...*/	Designates that the string following is a comment only and not an executable part of the code.
DECLARE	Causes the compiler to recognize calls to subroutines or user-defined functions.
END	If not part of a multiline statement (such as IF ... THEN), tells the compiler to stop execution at that point.
EQUATE	Establishes an equivalency between a constant and a value or between a constant and an expression (example: EQUATE BEEP TO CHAR(7)).
EXPENDABLE	Directs the compiler to create subroutine or function object code that will be cleared from memory when the program terminates.
FUNCTION	Directs the compiler to tag the object code of a program as being a user-defined function.
LINEMARK	Inserts a special character at this point in the object code which will permit the debugger to break in. This command is only used when compiling programs with the S option.
SUBROUTINE	Directs the compiler to tag the object code of a program as being an external subroutine.

Condition Logic

Condition logic commands enable a program to test the value of variables or expressions and then branch to appropriate routines.

- CASE** Tests a series of conditions (cases). When any one is met, the statements dependent on that condition are executed. All subsequent condition tests (cases) are skipped.
- IF...THEN...ELSE** Performs single line or multiple line logic. If not single-line, statements for both branches must be delimited with END. Multiline condition logic syntax is applicable to all conditional situations (such as READ or LOCATE statements).
- NULL** Used if a statement is called for, but no action is desired (example: IF ... THEN NULL ELSE ...).
- ON GOSUB** Branches to local subroutines. When the subroutine is terminated, control returns to the statement following this one.
- ON GOTO** Branches to one of several statement labels.
- UNTIL/WHILE** Establishes exit conditions for a loop created by the LOOP ... REPEAT or the FOR ... NEXT statements.

Device I/O

Device input and output (I/O) commands allow you to capture keyboard input or to print to a screen or to a printer. Other commands allow direct access to operating system ports.

- @** Establishes column and row coordinates at which text or data should be printed on the screen. The @() function can also be used for specific screen controls (clear screen, line, etc.).
- DATA** Puts data into the keyboard buffer. Data in the keyboard buffer is available to the next INPUT statement as if it had been typed in by an operator.
- INP()** Reads one byte from a specified MS-DOS port address.
- INPUT** Inputs data from a console or from a DATA statement into a variable. INPUT can be configured to accept a set number of characters, or to accept function and cursor control key input.
- OUT** Writes one byte to a specified MS-DOS port address.

PRINT Outputs data to screen or printer. Direct screen addressing (X,Y coordinates) are accessible via the @ function (example: @(10,2) prints at column 10, row 2). Printing to the printer is toggled by the PRINTER statement before printing begins.

Dictionary Interface

Dictionary interface commands allow you to access data on file via the dictionary from within an R/BASIC program.

{ } Retrieves the value of a field from the current record. The current record is defined as that stored in the variable @RECORD, with the item id stored in @ID. The dictionary being accessed must previously have been opened to the dictionary variable @DICT.

CALCULATE() Like the { } (braces) function, but can accept a variable in place of a field name. This permits a program to prompt for or otherwise determine which field to look up at execution time. Also requires access to @RECORD, @ID, and @DICT.

Dynamic Arrays

Commands listed in this section enable a program to manipulate data in Advanced Revelation format. For example, they allow you to add, delete, change, find, or sum data within the structure of Advanced Revelation's records (dynamic arrays).

< > Extracts or replaces a single element of a dynamic array

DELETE() Deletes a field, value or subvalue from a dynamic array.

EXTRACT() Fetches a field, value, or subvalue out of a dynamic array.

FIELD() Extracts a substring expression using a specified delimiter character and position.

FIELDSTORE() Replaces, deletes, or inserts a substring into a larger string, using any specified delimiter.

INSERT() Inserts a string as a new field, value, or subvalue in a dynamic array.

LOCATE, LOCATE BY Locates the position of a field, value or subvalue in a string, then returns the value of that position. Using BY allows you to sort an array.

REMOVE Fetches a field, value, or subvalue out of a dynamic array.

- REPLACE()** Replaces a field, value, or subvalue in a dynamic array with a designated string.
- SUM()** Sums the numeric values of the lowest level of a dynamic array, creating out of them a higher level (for example, summed subvalues become a value, summed values become a field).

Environment

Environment commands provide a means for an R/BASIC program to report on the environment within which it is being run, including time, date, and Advanced Revelation serial number. These commands also may change such environmental factors as printer settings and prompt characters.

- BREAK** Activates or deactivates the Break key.
- DATA** Loads input buffer. The next process(es) to require keyboard input will accept data from a DATA statement. DATA is not cleared when calling other programs or when the program is terminated.
- DATE()** Returns the current date in internal format.
- DRIVE()** Returns the current operating system drive and path.
- ECHO** Toggles the echo of input onto the screen as data is input using an INPUT statement.
- FLUSH** Forces the system to write all changed buffers to disk.
- GARBAGECOLLECT** Frees unused string space.
- GETBREAK()** Returns the value of the current break key, echo, printer, and prompt character settings.
- GETECHO()**
- GETPRINTER()**
- GETPROMPT()**
- GET.CURSOR()** Returns information about the current location and shape of the cursor on the video screen.
- PRINTER** Toggles output to a printer. While PRINTER ON is in effect, all PRINT statements send output directly to a printer.
- PROMPT** Sets prompt character produced by INPUT statements.
- PUT.CURSOR()** Sets the position and shape of the cursor on the video screen.
- SERIAL()** Returns the current Advanced Revelation serial number.

- STATUS()** Returns a number indicating the status or flag value of certain previously executed commands.
- TIME()** Returns the current time in internal format.
- TIMEDATE()** Returns the current time and date in external format.

File I/O

File input and output (I/O) commands allow you to read from Advanced Revelation data files or to write data into them.

- CLEARSELECT** Clears the contents of a select list (cursor).
- CLEARFILE** Deletes all records from a file but leaves the file definition.
- DELETE** Deletes a record from an opened file.
- FLUSH** Writes all changed buffers from RAM to disk (normally done at program termination only).
- HASH()** Returns the group number into which a record key will hash. This function applies only to Advanced Revelation ROS files.
- INMAT()** If used after an OPEN statement, returns the modulo of the file opened. If used after a MATREAD statement, returns the number of elements actually loaded by the read operation.
- MATPARSE** Turns a dymic array into a dimensioned array.
- MATREAD** Reads a full record from an opened file into a predimensioned array.
- MATUNPARSE()** Turns a dimensioned array into a dynamic array.
- MATWRITE** Writes an entire array onto a record in an opened file.
- OPEN** Opens a file to an internal file variable. Files do not need to be closed. OPEN statements open the data dictionary portions of a file separately, so that two open statements are required to open both portions. All file I/O (except XLATE) will use the internal file variable.
- READ** Reads a full record from an opened file into a variable.
- READNEXT, READNEXT BY** Reads the next record key from a selected list (cursor) into a variable. The BY option allows you to read the list bidirectionally.

READO	Reads a full record; if the filing system caches or buffers records, the read is fulfilled from the cache.
READV	Reads a single field into a variable from a record in an opened file.
SELECT, SELECT BY	Makes each record key in a file available to the READNEXT statement in a select list. The BY option allows you to access multiple select lists (cursors), and to specify sort criteria for the key list.
WRITE	Writes a full record onto an opened file.
WRITEV	Writes a single field onto a record in an opened file.
XLATE()	Extracts the value of a particular field in a particular record on file (opens file independently).

Iteration

Iteration commands provide a structured method to repeat execution of certain portions of the program.

FOR...NEXT	Loops a fixed number of times while the counter is less than or equal to a specified value. Can be used with UNTIL/WHILE.
LOOP...REPEAT	Repeats statements between the LOOP and REPEAT until some condition forces an exit. LOOP... REPEAT is frequently used with UNTIL/WHILE.

Logical

Logical commands determine whether certain expressions are logically true (returning a 1) or false (returning a 0).

AND and OR	Allows you to return a true or false value based on the Boolean AND or OR combination of two expressions.
ALPHA()	Returns a logical true if an expression evaluates to non-numeric.
BITAND(), BITOR(), BITXOR(), BITNOT()	Returns the logical AND, OR, XOR, and NOT of individual bits in a byte.
NOT()	Inverts the logical value of an expression (i.e., if an expression is true, the result after NOT will be false).
NUM()	Returns a logical true if an expression evaluates to numeric.

Networking

Networking commands enable an R/BASIC program to set and clear record locks in a networking environment.

LOCK Sets a lock semaphore in the network. If such a lock has already been set by another user, branches to ELSE logic.

☞ A **READ** will not check a **LOCK**. Only a **LOCK** can determine whether a lock has been set.

UNLOCK Clears a semaphore previously set by a **LOCK** statement.

Numeric Operations

Numeric operation commands deal with the arithmetic, algebraic and trigonometric manipulation of numbers.

ABS() Returns the absolute (unsigned) numeric value of a real number.

ALPHA() Returns a logical true if an expression evaluates to non-numeric.

ATAN() Returns the arctangent of an angle.

COS() Returns the cosine of an angle.

EXP() Returns the result of base e raised to the power of an expression.

INITRND Establishes a seed value for the **RND()** function.

INT() Returns the integer portion of a number.

LN() Returns the natural logarithm of a number.

MOD() Returns the modulo (remainder) of the division of two numbers.

NEG() Returns the inverse of a specified number

NUM() Returns a logical true if an expression evaluates to numeric.

PWR() Returns the value of a number raised to a specified power.

REM() Synonym for **MOD()**.

RND() Returns a randomly generated number.

- SIN()** Returns the sine of an angle.
- SQRT()** Returns the square root of a positive number.
- TAN()** Returns the tangent of an angle in degrees.

Operating System File I/O

Operating system file input and output (I/O) commands provide direct access to operating system files. They include the capability to create, delete, and change data in these files. The STATUS() function can be used after these commands to report on the success or failure of their execution.

- DIR()** Return directory information about an operating system file.
- DIRLIST()** Return an array of operating system file names on the path established with INITDIR.
- INITDIR** Establish a path and file mask for use with DIRLIST().
- OSBREAD** Reads a section of an operating system file into a variable. The programmer can specify the size of the section to be read, as well as the location in the operating system file from which the data is to be read.
- OSBWRITE** Writes data from a variable into a section of an operating system file. The programmer can specify the location in the operating system file to which the data is to be written.
- OSCLOSE** Closes an operating system file opened with a previous OSOPEN file.
- OSDELETE** Deletes an operating system file.
- OSOPEN** Opens an operating system file for use with the OSBREAD and OSBWRITE statements. Note that unlike Advanced Revelation files, operating system files must be closed (with the OSCLOSE command).
- OSREAD** Reads the entire contents of an operating system file into a variable. The file cannot exceed 64K in length.
- OSWRITE** Writes the contents of a variable to a operating system file. If the file exists, it will be overwritten. If it does not exist, it will be created.

Program Control

Although execution of a program generally proceeds from top to bottom, program control statements enable you to channel a program's execution to different sections, to subroutines, or to the debugger. Program control statements can also halt execution of a program and return it to menus and to TCL.

ABORT	Stops execution of program and returns directly to TCL.
CALL	Branches to an external subroutine.
CHAIN	Stops the current program and branches to another program or TCL command.
DEBUG	Halts program and calls the dynamic debugger.
END	Terminates a THEN or ELSE condition or functions the same as STOP.
GOSUB	Branches to an internal subroutine.
GOTO	Branches unconditionally to another portion of the program.
NULL	Used if a statement is called for, but no action is desired (example: IF ... THEN NULL ELSE ...).
ON GOSUB	Branches to local subroutines. When the subroutine is terminated, control returns to the statement following this one.
ON GOTO	Branches to one of several statement labels.
RETURN, RETURN TO	Branches back to calling statement + 1. If used from a function, RETURN can pass a value back to the calling routine. In an internal subroutine, TO allows you to specify a label as the return point.
STOP	Stops execution of the program and returns to the calling level of Advanced Revelation (the menu, TCL, etc.).

Report Formatting

Report formatting commands automatically handle paging logic, print headings and footings, and print column headings.

- COLHEADING** Creates a set of column headings to be printed at each page break.
- COLLENGTH** Establishes the width of columns being printed in the heading. **COLLENGTH** is used in conjunction with **COLHEADING**.
- FOOTING** Creates text to be printed at the bottom of each page. **FOOTING** initializes auto paging.
- HEADING** Creates a text heading which will be printed at the top of each new page. **HEADING** also initializes automatic paging.
- PAGE** Forces a page break.

String Formatting

String formatting commands allow a program to format data for output.

- FMT()** Formats text according to length and justification specified, and/or any masks.
- ICONV()** Converts from internal to external format or vice versa. Formats supported: boolean, date, datetime, decimal numbers, time, hex, scientific notation, and binary.
- CONV()** Converts from external to internal format. Formats supported are the same as **ICONV**.
- QUOTE()** Puts a specified string into double quotes (for example, to pass as literal text into an **R/LIST** heading).
- SPACE()** Generates a string of spaces.
- STR()** Creates a string of repeating characters.
- TRIM()**, **TRIMF()**, **TRIMB()** Deletes extra (more than one together) spaces in a string. **TRIMF** and **TRIMB** trim leading and trailing spaces, respectively.

String Manipulation

String manipulation commands allow you to find, add, change, delete, or count characters in a variable.

[n,m]	Extracts or assigns <i>n</i> characters starting at column <i>m</i> from or to a string.
CHAR()	Converts an ASCII sequence to its corresponding character.
COL1(), COL2()	Returns the column positions of delimiters used by the FIELD and bracket ([]) commands.
CONVERT	Converts a character in a string into another character.
COUNT()	Counts repetitions of a particular character in a string.
FIELD()	Returns a substring which precedes a defined delimiter.
INDEX()	Returns starting column location of a designated substring.
LEN()	Returns length of a specified string.
LOCATE	Finds position of a delimited substring in a larger string and return the position. The BY clause can also be used to sort items in a string.
SEQ()	Returns the ASCII sequence of a character.
SPACE()	Generates a string of spaces.
SWAP	Exchanges all occurrences of one substring in a string for another.

Subroutines

Subroutine commands provide the means to modularize programs. They allow you to establish local or external subroutines, to share data between routines, and to pass control between a main program and its modules.

CALL	Branches to external subroutine. External subroutines must be cataloged, and must have SUBROUTINE <i>prog.name</i> in the first line. Parameters can be passed to a CALL routine directly in the CALL, or through a COMMON area. CALL returns to the statement following after the external routine executes a RETURN.
COMMON	Sets up a common data area for sharing variables between external routines.

DECLARE	Directs the compiler to recognize a call to a user-defined function. Undeclared functions will cause the compiler to abort.
EXPENDABLE	Directs the compiler to create subroutine or function object code that will be cleared from memory when the program terminates.
FUNCTION	Establishes program as a user-defined function. This statement must be the first statement of the function.
GOSUB	Branches to an internal subroutine. Returns to the statement following when subroutine terminates with RETURN statement.
RETURN, RETURN TO	Branches back to calling statement + 1. If used from a function, RETURN can pass a value back to the calling routine. In an internal subroutine, TO allows you to specify a label as the return point.
SUBROUTINE	Establishes a program as a callable external subroutine. Prevents re-initialization of shared variables.

TCL/Operating System Command Interface

TCL/Operating System Command Interface statements provide R/BASIC with the capability to execute commands as if they had been entered at The Command Level (TCL) or at the operating system prompt. In some cases, such commands executed from within an R/BASIC program can pass data back into the program.

EXECUTE	Any statement in quotes following EXECUTE is executed as if it had been entered at TCL. EXECUTE does not save the results of the statement. To run a statement and then use the results in, for example, a subsequent READNEXT statement, see PERFORM.
PCPERFORM	Execute the operating system command that follows in quotes. In order for this to work, Advanced Revelation must be able to access the operating system command processor (for example, COMMAND.COM).
PERFORM	As with EXECUTE, any statement in quotes following PERFORM is executed as if it had been entered at TCL. PERFORM, however, makes the results available to a subsequent statement.
SUSPEND	Temporarily suspends Advanced Revelation, allowing you to execute operating system commands.

Variable Initialization

Variable initialization commands allow you to assign values to variables within an R/BASIC program and to create specialized variables. Specialized variables include dimensioned arrays and common areas to be shared by several routines.

CLEAR	Initializes all local variables in a program to null.
CLEARCOMMON	Initializes all common variables in a program to null.
CLEARDATA	Clears information stored by the DATA statement in the keyboard buffer.
CLEARSELECT	Clears a current select list (cursor).
COMMON	Establishes a variable which can be shared between a main program and external subroutines.
Labeled COMMON	Establishes a COMMON block which can be shared between any programs or subroutines. Labeled COMMON areas are not cleared when a program terminates, thus effectively serving as global COMMON statements.
DATA	Puts data into the keyboard buffer. Data in the keyboard buffer is available to the next INPUT statement as if it had been typed in by an operator.
DIMENSION	Creates a dimensioned array (1 or 2 dimensions).
EQUATE	Establishes an equivalency between a name and a constant value (EQU BEEP TO CHAR(7)) or another variable.
INITRND	Establishes a seed value for the random number generator (the RND() function).
MAT	Writes a value to all elements of a dimensioned array.
TRANSFER	Move the contents of a variable to another by reference.



Chapter N 3

Programming with R/BASIC

Before beginning to program in R/BASIC, review this chapter. Here you will find information about all aspects of R/BASIC programming. This includes fundamental concepts such as creating and calling your R/BASIC program, variables and literals, and the use of operators. More advanced topics include file access, arrays and matrices, and special system variables.

You do not need to master the contents of this chapter before you start writing R/BASIC programs. Instead, learn the basics and apply them to your programming requirements. You can then learn additional concepts as you need to. As you use R/BASIC commands and functions, you will be referred back to this chapter for the background information that is required for that command.

The Form of an R/BASIC Program	N 3 . 1
Variables	N 3 . 2
Constants	N 3 . 3
Null and Zero	N 3 . 5
Arithmetic Expressions	N 3 . 6
Arithmetic Operators	N 3 . 8
Comparison Expressions	N 3 . 9
String Operators	N 3 . 11
Statement Identifier Labels	N 3 . 12
Conditional Branching	N 3 . 14
Dynamic Arrays	N 3 . 15
Matrices	N 3 . 22
Subroutines	N 3 . 23
Internal Functions	N 3 . 27
External Functions	N 3 . 28
Accessing Advanced Revelation Files	N 3 . 29
Accessing Operator System Files	N 3 . 37

Cursors	N 3 . 40
The R/BASIC Debugger	N 3 . 44
Interfacing Dictionaries and R/BASIC	N 3 . 50
Special R/BASIC Variables	N 3 . 52
AREV.COMMON	N 3 . 60

The Form of an R/BASIC Program

The discussion below provides a general overview of the construction of R/BASIC programs.

Storing R/BASIC Programs

Each R/BASIC program is stored as one record in a file. The source code for each program can be up to about 34K long. Any number of programs can be stored in a file.

When a program is compiled, two additional records are generated. The object code record is stored in the same file, using the program name preceded with a dollar sign (\$). For example, the object code version of the program POST.INVOICES is called \$POST.INVOICES.

Another record that the compiler generates is a table of variables (a “symbol table”) for the program. This record has the same name as the program, but is preceded with an asterisk (“*”). The symbol table is used primarily by the R/BASIC debugger for tracing the values in variables.

Statements in R/BASIC Programs

The program itself consists of a series of statements or commands. R/BASIC statements are written on a single line. You cannot divide a statement across multiple lines unless there is a special provision for doing so (such as THEN and ELSE blocks). You do not have to terminate the statement on a line — the line feed at the end of each line is sufficient to signify the end of a statement.

More than one statement may be written on one line of a program, but each statement on a single line must be separated by semicolons. You can insert any number of blank lines and optional statement labels into the program to improve its readability.

You can also insert any number of spaces in front of or between the keywords, variables, and operators in a statement. For example, you can indent statements for clarity. There must be at least one space between keywords and variables. Operators (such as “=” and “+”) do not require spaces around them.

Statements can be entered in either lower case or upper case, as can variables. However, this does not mean that R/BASIC is case sensitive. All statement, functions, and variables are converted to upper case in a pre-compilation phase.

Variables

Variable identifiers refer to data that may change in value during the execution of a program. They permit accessing, processing, and storing of the data throughout the program.

Letter {letters, numbers, dollar signs, periods}

Letters, Numbers, Dollar Signs, Periods

A variable identifier, also known as a variable, reserves space for a value that may change dynamically during the execution of a program. It denotes that space throughout the program, and while the value of the variable can change, the variable name may not be changed.

The name must begin with an alphabetic character, but it may then contain letters, numbers, dollar signs, or periods. Spaces and minus signs (-) within a name are not allowed. Variable names can appear in either upper or lower case letter. However, names are not case-sensitive. All names are converted to upper case during compilation.

A variable may have either a numeric value or a string value and its value or length may change dynamically during execution of the program. The value of a variable identifier may also change from numeric to string, or from string to numeric during the course of a program. For example, the variable identifier CKBAL may be assigned the value "Checkbook Balance" at the beginning of a program and may end the program with the value \$110.95.

☞ R/BASIC keywords, statements, and functions should not be used as variable identifiers.

Value Assignment

Many R/BASIC statements, such as CLEAR COMMON, CLEAR, and MAT, can assign values to variable identifiers in a program. A variable identifier must be assigned a value by an assignment statement in the program before it can be used. An unassigned variable will cause an error message when run. Storage space for variables is allocated in the order in which the variables occur. A slight reduction in program execution time is possible through assigning the most commonly used variables first.

Correct Use of Variable Identifiers

```
INPUT DEPOSIT
INPUT CHECKS
CKBAL ='Checkbook Balance '
CKBAL = CKBAL : DEPOSIT - CHECKS
PRINT CKBAL
```

The final contents of the variable CKBAL is printed as Checkbook Balance followed by the calculated amount.

Incorrect Use of Variable Identifiers

`1ST = 1`

The variable identifier does not begin with a letter.

`COMMON BACK-UP`

A dash is used as part of the variable.

`SEPT, 82 = DATE`

A comma is used as part of the variable.

`PLAT NO`

The variable is not part of an expression or statement.

Constants

Constants are numbers or character strings that keep their value throughout the execution of a program. Constants do not change value, data type, or length during a program.

Numeric Constants

A numeric constant is any number, including scientific notation, that represents a quantity. Numeric constants are any number, positive or negative, up to 18 digits long, including fractional digits. A unary minus sign (-) denotes a negative number. Numeric constants may not include commas or dollar signs. A numeric constant does not require quotation marks.

```
...n...
...nE{+}{-}n...
...ne{+}{-}n...
```

Scientific Notation

Scientific notation is a number followed by an upper case or lower case E, a plus (+) or minus (-) sign, and by as many as four exponential digits, in that order. Scientific notation may be used at input or whenever numeric constants are designated.

Numeric Constant Limits

R/BASIC supports up to 18 digits of precision (64 bits). Numbers of any arbitrary size are accepted if you enter them in scientific format (up to the calculation limits listed below).

You can input numbers of any size, but are limited to 18 places to the right of the decimal point.

Calculation Limitations R/BASIC supports calculations to these sizes:

Positive: +10**-4932 to +10**+4932

Negative: -10**-4932 to -10**+4932

Character String Constants

A string constant is a sequence of characters which may include letters, numbers, dollar signs, or other keyboard characters. It may be up to 254 characters.

String constants must be enclosed in single quotation marks ('aaa') or double quotation marks ("aaa").

```
"{characters}"
'{characters}'
```

If a set of quotation marks is to be part of the actual string, use the other set of quotation marks as delimiters. For example:

```
"This is a valid 'string' constant"
```

```
'This also is a valid "string" constant'
```

A null constant can be denoted by two quotation marks with no characters between them: ("") or (").

- ☞ Some keyboards have a key (') that looks similar to the beginning single quotation mark. Do not use this key for a quote mark. Advanced Revelation will not recognize it as a quotation mark.

String Size Limits The longest string allowed during program execution is 65,530 characters. The longest string allowed in quotes in an R/BASIC statement is 254 characters long.

Hex Constants

To include non-printing (or other) characters using their hex value, enclose the hex representation of the characters with backslashes (\). For example:

```
\0D0A\
```

```
\0D0A\
\FE\
```

The first example is the hex constant that represents a carriage return-line feed combination. The second example is the hex constant representing a field mark (ASCII 254).

Correct Use of Constants

987654321.1234

A numeric constant.

3.14159

A numeric constant.

-9.7E3

The number 9700 in scientific notation.

"FINISHED"

A character string constant.

'23 Main Street'

A character string constant.

""

A null character string constant.

\4142434445\

The hex representation of the string "ABCDE"

Incorrect Use of Constants

A

A is a variable since it has no quote marks.

"DATE

The terminating quotation mark is missing.

"JOE DOE'

Quotation marks are mismatched.

/FDFF/

A slash is used, not a backslash.

Null and Zero

Null and 0 (zero) are given special consideration in R/BASIC.

In arithmetic statements, null is treated as a numeric zero. A null is considered to have a logical value of false. If a null is compared to a numeric zero, they will not be equal; the null is less than 0 (zero).

This feature can cause problems in applications which use dynamic arrays to store numeric data. If a null is stored in place of a 0 (zero), it will not compare equal if extracted. For example:

```
MTH = 2
AMTS = '100'
THIS.MTH = AMTS<MTH>
IF THIS.AMT EQ 0 THEN PRINT 'THIS.MTH IS ZERO'
```

THIS.MTH is set to null because there is no second element in the string in AMTS. This code will not print 'THIS.MTH IS ZERO' even though the variable THIS.MTH is empty. The last line could be changed to read:

```
IF THIS.MTH + 0 EQ THEN PRINT 'THIS.MTH + 0 YIELDS ZERO'
```

The expression THIS.MTH + 0 yields a 0 (zero).

Correct Use of Null and Zero

```
N = ''
```

N will have a value of null.

```
IF N THEN Y = 1 ELSE Y = 0
```

Y will have a value of 0 because N is null.

```
Z = 5 + N
```

Z will have a value of 5 because N is null.

```
IF N EQ 0 THEN X = 1 ELSE X = 2
```

X will have a value of 2 because N is null.

```
A = (N LT 0)
```

A will have a value of 1 because N is null.

```
B = NUM('')
```

B will have a value of 1.

```
C = ALPHA('')
```

C will have a value of 0.

Arithmetic Expressions

Arithmetic expressions are combinations of variables, constants, or functions that specify a rule to calculate a single numeric value.

```
expression {operator expression ... }
```

Simple Arithmetic Expressions

The simplest arithmetic expressions consist of a single numeric constant, a variable, or an intrinsic function. Following are examples of simple arithmetic expressions:

```
3.1416    1000    .0005
QTA      INT(2.5) "144"
```

An arithmetic operator (+, /, *, (), etc.) can combine two arithmetic expressions and produce a third arithmetic expression. The third arithmetic expression may in turn be combined with other expressions. For example:

```
BAL.FORD + MO.END
SCOREA + SCOREB/2
AMT*.065
SIN(D)/COS(D)
```

Arithmetic operations follow a strict order of priority. See the "Priority of Arithmetic Operators" table under "Arithmetic Operators."

R/BASIC considers character string values that contain only numeric characters to be decimal numbers. For example, the expression 123 + "321" will evaluate to 444.

Character string values that contain non-numeric characters will produce an error message when the program is run. The string value will assume the value of 0 (zero) in this case and the program will terminate.

Correct Use of Arithmetic Expressions

```
RESULT=-6+3
```

RESULT will be -3.

```
RESULT=- (6+3)
```

RESULT will be -9.

```
RESULT=3*4**2+A
```

RESULT will be 4 raised to the second power (16), times 3 (48), plus the current value of A.

Incorrect Use of Arithmetic Expressions

```
18 + 24/6
```

The expression is not part of a statement.

```
NAME = 'EARL' + 1
```

Non-numeric characters are used in the expression.

Arithmetic Operators

Arithmetic operators form arithmetic expressions which R/BASIC evaluates according to a strict order of priority. The result is always a numeric value.

```
expression {operator expression ... }
```

Valid Arithmetic Operators

The valid arithmetic operators in R/BASIC are listed following. Note the keyboard symbols that are used to signify each operation. In particular, note that one asterisk (*) signifies multiplication while two asterisks (**) signify exponentiation. Division is indicated by a slash mark (/).

An expression may have more than one operator. Each operator has a priority rating, as shown in the chart following. R/BASIC scans the entire expression and begins the evaluation with the operator that has the highest priority. If two or more operators have the same priority in a given expression, evaluation proceeds from left to right.

Multivalued Operators

The four operators +++, —, ***, and /// allow addition, subtraction, multiplication, and division of two multivalued or subvalued strings.

For example, if +++ is used to add two multivalued strings, each value in the first string is added to the value in the corresponding position in the second string. If the strings are of unequal length, Advanced Revelation will add 0 (zero) to the unmatched positions.

For example:

```
A = 12121212
B = 222222222222
PRINT A+++B
```

The result will be 3²3²3²3²22. Zero (0) is added to the fifth multivalue.

Parentheses

Parentheses may be used to alter the order of priority. Expressions that are enclosed in parentheses are always evaluated before the expressions outside the parentheses. If parentheses are nested, as in this example:

```
(3*(3+3))
```

the innermost expression is evaluated first.

Parentheses may also be used anywhere in an expression to clarify its order or to increase readability of the program.

Priority of Arithmetic Operators

Priority	Operation	Keyboard Symbol
high 1	unary plus	+
1	unary minus	-
2	exponentiation	**
2	exponentiation	^
3	multivalued multiplication	***
3	multivalued division	///
3	multiplication	*
3	division	/
4	multivalued addition	+++
4	multivalued subtraction	---
4	addition	+
4	subtraction	-
5	multivalued concatenation	:::
low 5	concatenation	:

Correct Use of Arithmetic Expressions

24/2*4

Result will be 48. Multiplication and division have the same priority.

24/(2*4)

Result will be 3. The expression within the parentheses is evaluated first.

Comparison Expressions

Comparison expressions compare two arithmetic or string expressions. Comparison expressions always evaluate to either 1 (one), if true, or 0 (zero), if false.

expression {comparison.operator} expression ... }

Comparison Operators

A comparison operator compares the values of two expressions and returns that evaluation as true or false. True evaluations yield a 1 (one) and false evaluations a 0 (zero). Expressions that are compared may be either arithmetic or string expressions. Arithmetic expressions compare the values of the two operands. The chart following lists the valid R/BASIC comparison operators.

Comparison expressions are evaluated from left to right; each pair of expressions is compared and reduced to a value of 1 (one) or 0 (zero) before the next specified operation is executed.

String Comparisons

String comparisons compare the characters in corresponding positions in each string, starting with the leftmost characters. If each pair of characters matches the *comparison.operator* specification, the expression will evaluate to 1 (true). If the strings are identical up to a point, but one string is longer, the longer string is judged to be greater.

In string comparisons, the string with the character that has the higher numbered ASCII code equivalent will be judged to be of greater value. For example, 'Olsen' > 'Olson' would evaluate to 0 (false) because **o** in Olson is judged to be greater than **e** in Olsen when the strings are compared letter for letter.

A 0 (zero) is evaluated to be greater than a space; a character string is evaluated to be greater than a null string. Comparison operators have a lower priority than all arithmetic and string operators and are evaluated after all arithmetic and string operations have been executed.

Comparison Operators

Operator Symbol	Operation
EQ =	Equal to
NE # >< <>	Not equal to
GT >	Greater than
LT <	Less than
GE >= =>	Greater than or equal to
LE <= <=	Less than or equal to
MATCH or MATCHES	Pattern matching

Correct Use of Comparison Operators

A = P < 500

Evaluates to 0 (false) if the current value of P is less than 500.

B = 12345 GT 12345

Evaluates to 0 (false).

C = "bric" > "brac"

Evaluates to 1 (true).

String Operators

R/BASIC provides two operators that manipulate strings of characters: the concatenation operator (:) to join strings, and the [] (bracket) operator to extract strings.

```
string : string
string CAT string
variable[start, length]
```

Joining Strings

A concatenation operator can create a string by joining any number of smaller strings. For example:

```
CITY = 'Seattle'
STATE = 'Washington'
ZIP = '98188'
PRINT CITY: ' ', ':STATE: ' ':ZIP
```

will output:

```
Seattle, Washington 98188
```

Concatenation

Note that the *Priority of Arithmetic Operators* table under “Arithmetic Operators” shows concatenation as the lowest priority. Therefore, if a concatenation operation is given in an expression with any arithmetic operation, the concatenation will be performed last.

Brackets

Use the [] (bracket) function to select small portions of a large string. This is particularly useful to output data in a meaningful form after storing it in a tight pattern to save storage space. A phone number stored as 4062269362 could be printed in a familiar pattern using the bracket function. For example

```
NN = "4062269362"
PHONE = "(" : NN[1, 3] : ")" : NN[4, 3] : "-" : NN[7, 4]
PRINT PHONE
```

will print:

```
(406) 226-9362
```

For more information, see [] (bracket) in the chapter “R/BASIC Command Reference.”

Correct Use of String Operators

```
SHORT = "538006342"
SSN = SHORT [1,3] : '-' :SHORT [4,2] : '-' :SHORT [6,4]
PRINT SSN
```

The output would be 538-00-6342 in the familiar pattern of Social Security numbers.

```
PRINT "RECORD READ AT " : TIMEDATE()
```

RECORD READ AT would be printed with the current internal system time and date concatenated.

Incorrect Use of String Operators

```
Z = C : : Q
```

An extra colon is used.

```
A = B: 'AT' :
```

A colon is used at the end of the assignment statement.

Statement Identifier Labels

Statement identifier labels are optional names or numbers used for reference in a program. They may also be used to improve its readability. Any R/BASIC statement may begin with a statement identifier label.

```
numerics(s){:} {statements}
letter{alphanumeric(s)}: {statements}
```

Label Characters

Labels may be numeric or alphanumeric. Labels beginning with a letter may be followed by other letters, numbers, periods, or dollar signs. They may be of any length.

Colons

A colon following a numeric label is optional. A colon following an alphanumeric label is mandatory. Using colons even when they are not required increases readability and maintains consistency.

Blank Spaces

Do not use blank spaces within a label. Blank spaces will cause compilation errors.

Statement identifier labels or label numbers may be used or omitted at your discretion, but they must be used for reference in a program in conjunction with the GOSUB, GOTO, and RETURN TO statements.

A statement identifier label may stand alone on a line.

- ☛ Although Advanced Revelation accepts statement identifier labels with the same name as variables in a program, it is not a recommended programming practice.
- ☛ Important: Spacing is critical when referring to a statement identifier label in a program.

For example:

```
GOTO 300:
```

would be read by R/BASIC to mean go to label 300, while the same statement without a space

```
GOTO300:
```

would be taken as a new label.

Correct Use of Statement Labels

```
100:
```

Numeric label with colon

```
100
```

Numeric label

```
EX1:
```

Alphanumeric label with colon

```
PAGE:
```

Alpha label with colon

```
B$.30:
```

Alphanumeric label with dollar sign and period

Incorrect Use of Statement Labels

```
GOTO 1ST
```

Labels that begin with a number must be all numeric.

```
AA1 PRINT 'DATE'
```

An alphanumeric statement label must be followed by a colon.

```
TOTAL SALES:
```

A label cannot contain blanks.

Conditional Branching

Conditional branching refers to the ability of software to do different things based on a comparison of input data values or computations. A false result or value is zero or null. Any other result or value is true.

```
IF test THEN true.sequence ELSE false.sequence

IF test THEN
    statements
END ELSE
    statements
END

BEGIN CASE
    CASE test
        statements
    CASE test
        statements
    CASE test
        statements
END CASE
```

IF ... THEN ... ELSE

The IF ... THEN ... ELSE statements provide the vehicle for conditional branching. R/BASIC evaluates the truth or falseness of a specified test condition and transfers program control to the THEN sequence if the test condition evaluates to true, and to the ELSE sequence if the condition is false.

For more information, see IF in the chapter “R/BASIC Command Reference.”

BEGIN CASE ... END CASE

Where the IF statement tests for a condition that is one way or another, the CASE statement allows branching for more than two conditions. Each *test* is evaluated from the top down beginning at the first CASE. The statements for the first true *test* will be performed. Program control moves to the first statement following END CASE when the statements of the first true *test* are finished. For more information, see CASE in the chapter “R/BASIC Command Reference.”

Correct Use of Conditional Branching

```

IF X THEN GOTO 5 ELSE Y = 1

IF Y EQ X THEN Y = Z; Z = 1 ELSE
  PRINT 'DONE'
  STOP
END

IF END.OF.MO THEN
  IF (TODAY - PMT.DATE) > 30 THEN
    PRINT "OVERDUE  ":CUST.BAL
  END ELSE
    PRINT "BALANCE  ":CUST.BAL
  END
END

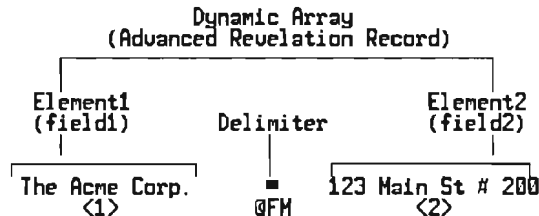
TRUE = 1
BEGIN CASE
  CASE SALES.DATE  FIRST.QTR
    FIRST.QTR.COM += COMMISSION
  CASE SALES.DATE  SECOND.QTR
    SECOND.QTR.COM += COMMISSION
  CASE SALES.DATE  THIRD.QTR
    THIRD.QTR.COM += COMMISSION
  CASE SALES.DATE  FOURTH.QTR
    FOURTH.QTR.COM += COMMISSION
  CASE TRUE
    MESSAGE = 'Incorrect Date'
    GOSUB ERROR
END CASE

```

Dynamic Arrays

Fundamental to Advanced Revelation architecture are dynamic arrays. The following section shows how dynamic arrays provide flexibility and straightforward solutions to complex problems which often absorb a significant amount of time and effort in other environments.

A dynamic array is a string of characters divided into one or more parts, or elements, by special characters called delimiters. The basic dynamic array has one delimiter which defines two elements.



Example of a Basic Dynamic Array

Because the length and number of elements in an Advanced Revelation dynamic array is flexible, you can easily change the length of fields within an Advanced Revelation database without redesigning the application. For example, in database applications which require fixed length fields, changing the length of a zip code from five to nine characters would involve complete restructuring of field lengths. Adding a new field would cause the same problem in the record. Because Advanced Revelation uses dynamic arrays, however, it can adapt to the length and number of elements and fields without any restructuring. You can simply enter a longer zip code in the field or add a new field to the record.

Dynamic Arrays and Records

A record is a dynamic array. The elements of a record are defined by special delimiters. Fields, values and subvalues are defined by field marks (ASCII character 254), value marks (ASCII character 253), and subvalue marks (ASCII character 252). A dynamic array can have three dimensions using these delimiters. R/BASIC provides the system variables @FM and @VM for use as a field mark and a value mark. Use the following statements to read and write dynamic arrays:

Statement	Description
READ	Assigns a record to a variable
WRITE	Writes a variable to a file
READV	Assigns a field in a record to a variable
WRITEV	Writes a variable to a field

☛ A variable in the descriptions above is a dynamic array when it contains one or more delimiters.

Subscripts

A dynamic array appears to R/BASIC as a string with delimiters. The delimiters define the elements of the array. An element is located by referencing it with a subscript. The example following illustrates the use of subscripts to access different elements. Note that @FM, @VM, and @SVM are used for field mark, value mark, and subvalue mark:

Array with 2 fields	
array:	"Field1":@FM:"Field2"
subscripts:	<1> <2>
Array with 3 fields (Field 2 has 2 values)	
array:	"Field1":@FM:"Value1":@VM:"Value2":@FM:"Field3"
subscripts:	<1> <2,1> <2,2> <3>
Array with 2 fields (Field 2 has 3 values; Value 2 has 2 subvalues)	
array:	"Field1":@FM:"Value1":@VM:"Subvalue1":@SVM:"Subvalue1":@VM:"Value2"
subscripts:	<1> <2,1> <2,2,1> <2,2,2> <3>

Using Dynamic Arrays

R/BASIC is designed for a database environment. This design is reflected in the statements and functions used with dynamic arrays. These statements and functions use the delimiters to perform a range of operations:

- Assigning elements to variables
- Locating strings in elements
- Adding elements
- Deleting elements
- Replacing the data in elements
- Sorting elements
- Performing operations using the elements of two arrays
- Converting delimiters to and from other characters

This section explains the use of dynamic arrays in each of these topics. The R/BASIC statements and functions used in these topics are fully documented elsewhere in the manual. The examples use the variable CUST.REC which is set to the contents of a record by the READ statement. Field 6 is a field which contains values and subvalues. Any variable which contains delimiters can be used as a dynamic array.

Assigning Elements to Variables

A dynamic array assignment copies part of an array to a variable without changing the array.

Assigning an Element to a Variable

The element (field) can contain values and subvalues. The example below assigns field 6 of CUST.REC to the variable

NAMES. When NAMES contains delimiters it can be used as an array within a program. Value and subvalue subscripts are not used because the entire field will be assigned.

```
NAMES = CUST.REC<6>
```

This is the short form of EXTRACT. It is equivalent to:

```
NAMES = EXTRACT(CUST.REC, 6, 0, 0)
```

Assigning Multiple Fields to a Variable

The variable is a dynamic array with field marks in addition to whatever value or subvalue marks it may contain. The example below assigns 2 fields, beginning at field 6, to the variable NAMES.TITLES. The 2 is not a subscript. The system variable @FM is used as the delimiter.

```
NAMES.DTS = FIELD(CUST.REC, @FM, 6, 2)
```

Assigning a value to a variable

A single value that is part of a field is assigned by using its subscript. A single subvalue in a value is assigned by using a subvalue subscript. The below example assigns one value to a variable.

```
OWNER = CUST.REC<6, 3>
```

An alternative syntax:

```
OWNER = EXTRACT(CUST.REC, 6, 3, 0)
```

Locating strings in elements

One element or the entire array may be searched for a string or value. The example searches field 6 for a value which contains MRS. If the value is found, POS is set to the value position number where MRS is located.

```
LOCATE 'MRS' IN CUST.REC<6> USING @VM ⇨  
SETTING POS ELSE NULL
```

Adding Elements

One or more elements may be added or deleted. New elements may be added at the end, at the front, or at a specific position in the array.

Adding Elements to the End of the Array

The example below assigns the contents of NAMELIST to the end of field 6 in CUST.REC. The -1 indicates the end of the array.

```
CUST.REC<6, -1> = NAMELIST
```

or

```
CUST.REC = REPLACE(CUST.REC, 6, -1, 0, NAMELIST)
```


Adding Elements to the Front of the Array

The example below assigns the contents of NAMELIST in front of the existing contents of field 6. The 0 specifies that the insert will come in front of other values in the field. The 0 is not a subscript.

```
CUST.REC = FIELDSTORE (CUST.REC, @FM, 6, 0,  =>
                        NAMELIST)
```

Adding Elements in the Middle of an Array

The example below inserts NAMELIST in front of the third value in field 6.

```
CUST.REC = INSERT (CUST.REC, 6, 3, 0, NAMELIST)
```

Adding a number of elements

Use a subscript larger than the number of elements in the array. For example, if CUST.REC had 8 fields (elements), then:

```
CUST.REC<14> = NAMELIST
```

adds elements 9 through 14 and assign NAMELIST to element 14.

Deleting Elements

One or more elements may be removed from an array.

Deleting One Element

The first value in field 6 is removed. The field now has one less value. For example:

```
CUST.REC = DELETE (CUST.REC, 6, 1, 0)
```

Deleting Multiple Elements

The example below deletes three values from field 6, beginning with value 2 and continuing through value 4. The statement uses the delimiter @VM and replaces values 2 through 4 with null. The 3 specifies the number of elements to delete and is not a subscript.

```
CUST.REC = FIELDSTORE (CUST.REC, @VM, 2, 3, '')
```

Replacing the Data in Elements

The contents of one or more elements can be replaced. Part of a dynamic array may be replaced by a value or another dynamic array.

Element

The example below assigns ALAN JEFFERSON to field 6, value 4:

```
CUST.REC<6, 4> = 'ALAN JEFFERSON'
```

In this example, value 2 in field 6 is replaced by the dynamic array NAMELIST.

```
CUST.REC<6,2> = NAMELIST
```

Replacing Multiple Elements

The next example replaces values 2 through 4 in field 6 with the dynamic array NAMELIST. The 3 specifies the number of elements to replace.

```
CUST.REC<6> =  
FIELDSTORE(CUST.REC<6>, @VM, 2, 3,  =>  
NAMELIST)
```

Sorting elements

The data in dynamic array elements may be sorted alphabetically or numerically in ascending or descending order.

Field 6 is sorted using ascending left justification. This example first locates the element where MEWS should be in the array. If MEWS is not in that element, then POS, which is set to the element number, is used by the INSERT function to insert MEWS in the correct position.

```
LOCATE 'MEWS' IN CUST.REC<6> BY 'AL' USING @VM  =>  
SETTING POS ELSE  
CUST.REC = INSERT(CUST.REC, 6, POS, 0, 'MEWS')  
END
```

Performing Operations with Multiple Arrays

Two arrays can be added, multiplied, divided, subtracted, and concatenated.

Multiplying Two Arrays

The following example is used to determine the extended price of line items on an invoice. The array QUANTITIES is a multivalued array of quantities for ordered parts. PRICES is a multivalued array of prices for each of the parts. The extended cost of one line item is reached by multiplying the price per item by the number of items. The example yields an array of extended prices.

```
EXT.PRICES = PRICES *** QUANTITIES
```

Addition, division, subtraction, and concatenation are performed in the same manner.

Converting Delimiters to Other Characters

Any character may be converted to a delimiter and a delimiter may be converted to another delimiter or any character.

In the following example, the record NAMELIST is a saved list of names. The names are separated by field marks. The example converts the field marks to value marks and assigns the array to a value in another array.

```
CONVERT @FM TO @VM IN NAMELIST
CUST.REC<6,1> = NAMELIST
```

Dynamic Arrays vs Matrices

Both a dynamic array and a matrix contain elements. The number of elements of a matrix are fixed by the DIMENSION statement. In contrast, the number of elements in a dynamic array may be changed by changing the number of delimiters.

A dynamic array is one variable and each element in a matrix is one variable. It follows that each element of a matrix may contain a dynamic array. Matrices are useful for breaking up long arrays to increase string searching speed. Dynamic arrays are useful when flexibility is important. Records may be read or written using matrices. For example:

MATREAD	assigns each field in a record to a variable which is a matrix element.
MATWRITE	writes each matrix element to a field in a record.

Using Matrices and Dynamic Arrays

A matrix may be converted to a dynamic array and a dynamic array may be converted to a matrix. The example parses each dynamic array field into one matrix element.

```
MATPARSE CUST.REC INTO CUST.MAT
```

The following example parses each matrix element into one dynamic array element:

```
CUST.REC = MATUNPARSE(CUST.MAT)
```

If a matrix element contains delimiters, dynamic array operations may be used. In each of the examples below the field mark subscript must be 1.

Assign a Value to a Variable The following example extracts a single value from a dynamic array within a matrix element, and assigns it to a variable:

```
OWNER = CUST.MAT(6)<1,1>
```

Add Elements to the End of the Array In this example, a value is assigned to the end of a multi-valued field within a matrix element:

```
CUST.MAT(6)<1,-1> = NAMELIST
```

Replace Multiple Elements

Note that FIELDSTORE uses 6 to indicate occurrence not subscript number:

```
CUST.MAT (6) =  
FIELDSTORE (CUST.MAT (6) , @VM, 2, 6,  =>  
NAMELIST)
```

For complex problems the maximum number of dimensions is five using standard subscripts. For example:

```
VAR = MAT1 (1, 2) <3, 4, 5>
```

For more information, see “Arithmetic Operators” elsewhere in this chapter. Also see <> (Angle Bracket Operators), CONVERT, DELETE, EXTRACT, FIELD, FIELDSTORE, LOCATE, MATPARSE and REPLACE in the chapter “R/BASIC Command Reference.”

Matrices

The word matrix is used to describe a variable which is tabular, that is, it has more than one data element, such as a table or an array.

```
matrix.variable (subscript)  
matrix.variable (row.subscript, column.subscript)
```

matrix.variable

A matrix is a variable with more than one associated data element. Matrices allow related information to be stored together so that data elements can be easily accessed and processed.

Each piece of data in the matrix is called an element of the matrix. Each element is identified by the numeric location at which it is stored. Therefore, any element can be accessed by specifying its location in the array. An element may contain a dynamic array.

The numeric location is specified by a subscript in parentheses following the matrix variable identifier. By general agreement, the first subscript always refers to the row or line in the array and the second subscript always refers to the column.

Subscripts

A matrix with one subscript, called a one-dimensional matrix, is actually a list of data elements, one element on each row or line.

A two-dimensional matrix has two subscripts, indicating rows and columns, so the elements are stored in two dimensions. Therefore, a matrix may be thought of as a table or array, that is, a rectangular arrangement having two dimensions, length

and width or rows and columns. Again, any element can be accessed by specifying its row and column position within the matrix.

Dimensions

Before a matrix can be used in an R/BASIC program, it must be named and its maximum dimensions must be specified, because R/BASIC must know how much storage space to set aside. Use the DIMENSION or COMMON statements at the beginning of the program to name the matrix and specify its dimensions. The total number of variables and array elements in a program should be less than 5,000.

Correct Use of Matrix

```
DIMENSION M(4,5)
```

The matrix M is dimensioned to 20 available elements as shown in the chart below.

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	M(1,1)	M(1,2)	M(1,3)	M(1,4)	M(1,5)
Row 2	M(2,1)	M(2,2)	M(2,3)	M(2,4)	M(2,5)
Row 3	M(3,1)	M(3,2)	M(3,3)	M(3,4)	M(3,5)
Row 4	M(4,1)	M(4,2)	M(4,3)	M(4,4)	M(4,5)

The matrix M(4,5) contains 21 elements, 4 by 5 plus one more, the 0 (zero) element M(0,0).

Subroutines

A subroutine is a separate program or a paragraph within the main program that handles unique functions or tasks that are needed by the calling program.

Subroutines

A subroutine is a sequence of instructions that performs a commonly used task. At any point in the calling program, control can be transferred from the calling program to a subroutine. When the subroutine statements have been executed, control returns to the program line following the line that called the subroutine.

Efficient use of subroutines makes writing programs easier. Large, complicated programs can be organized into manageable subroutines which are coded individually and which can be checked individually for accuracy.

Internal Subroutines

An internal subroutine is contained within the program which calls it. Control is passed from the main body of the program to the internal subroutine by a GOSUB

statement. GOSUB specifies the statement label which identifies the needed subroutine. A RETURN statement at the end of the subroutine returns control to the line following the GOSUB.

External Subroutines

An external subroutine is coded as a separate program. The external subroutine must have been previously compiled and cataloged before it can be used. Use the CALL statement to transfer control from a calling program to an external subroutine. A RETURN statement within the subroutine will return control to the calling program at the line after the call was made.

The first line of code in an external subroutine must be a SUBROUTINE statement. This SUBROUTINE statement specifies the name and arguments which other programs must use when entering or calling the subroutine.

Since an external subroutine is a separate entity, its statement labels and variables are its own, and they do not relate to the calling program's labels. An external subroutine may also contain its own internal subroutines, and it may call other external subroutines. It may also call itself (as a recursive subroutine). A RETURN statement is required to return control to the calling program.

An external function, a special type of external subroutine, is called as part of an expression. The function returns a value to the position in the expression from where it is called.

When either an external subroutine or function is called, the program remains in memory until the main program that called it terminates with a STOP statement. This is efficient if you intend to call the same subroutine or function several times, since the object code for the external routine does not need to be re-loaded each time.

However, the object code remaining in memory occupies space. If you wish to flush the object code out of memory as soon as the external subroutine quits (thereby conserving RAM), use the statement EXPENDABLE. This is less efficient if you must call the routine repeatedly, since the routine will need to be re-loaded each time you call it.

Variables

An external subroutine may contain a variable which has the same name as the main program. The variable may be changed in the subroutine without affecting a variable of the same name in the main program. The value of a variable may be shared by passing the variable in the CALL statement or specifying the variable in the COMMON statement.

Passing by Reference

Variables are passed to subroutines in the order in which they appear in the CALL statement. They are passed by reference.

Variables specified in a COMMON statement will be shared in the order in which they appear in the calling program. The subroutine may use different names to share the COMMON variables. A subroutine may also use different names for the variables which are passed. For example:

```
* main program
COMMON A,B,C
A = 1
B = 2
C = 3
CALL SUB
STOP

SUBROUTINE SUB
* subroutine
COMMON X,Y,Z
PRINT X:Y:Z
RETURN

* main program
A = 1
B = 2
C = 3
CALL SUB (A,B,C)
STOP

SUBROUTINE SUB (X,Y,Z)
* subroutine
PRINT X:Y:Z
RETURN
```

Both subroutines would print 123.

Passing by Value

The value of a variable may be made local to a subroutine by passing its value as an expression. Passing the variable as an expression allows the variable to be modified in a subroutine without changing its value in the calling program. For example:

```
* main program
A = 1
B = 2
C = 3
CALL SUB (A,"",B*1,C+0)
PRINT A:B:C
STOP
```

```

SUBROUTINE SUB(X,Y,Z)
* external subroutine
A = A + 1
B = B * A
C = C**B
PRINT C
RETURN

```

The subroutine would print 81 and the main program would print 123. An assignment to a local variable within the subroutine will achieve a similar effect.

For more information, see CALL, COMMON, DECLARE, EXPENDABLE, FUNCTION, GOSUB, MAKEVOC, and RETURN in the chapter "R/BASIC Command Reference."

Sample Program Using Internal and External Subroutines

```

ARGUMENT = ''
LOOP UNTIL ARGUMENT = 'END'
  GOSUB GETNUM
  IF VALID.NUM THEN GOSUB PRINTANSWER
REPEAT
STOP

GETNUM:
PRINT @(0,10):@(-4):'ENTER ARGUMENT ':
INPUT ARGUMENT
VALID.NUM = NUM(ARGUMENT)
IF VALID.NUM THEN
  CALL FACTORIAL(ARGUMENT,ANSWER)
END ELSE
  PRINT @(0,12):@(-4):"ARGUMENT OF '":ARGUMENT:"' IS NOT NUMERIC"
END
RETURN

PRINTANSWER:
PRINT @(0,12):@(-4):"FACTORIAL OF ":ARGUMENT:" = ":ANSWER
RETURN

SUBROUTINE FACTORIAL(X,A)
A = 1
FOR I = 1 TO X
  A = A * I
NEXT I
RETURN
END

```


Internal Functions

A function is a subroutine that returns one value to the place from where it was called. Certain commonly used functions are built into R/BASIC to facilitate programming.

Internal Functions

An internal function must be given as an element of an expression. The returned value of the function is evaluated as part of the expression. It may appear wherever an expression is valid.

An internal function depends upon specified arguments for its value. The arguments must be enclosed in parentheses and they must follow the function name. The enclosed arguments must be separated by commas. The parentheses must be complete with both the opening parenthesis "(" and the closing parenthesis ")".

☞ No spaces are allowed between the function name and the opening parenthesis; however, spaces are permitted between the arguments.

Parentheses

In functions such as TIMEDATE(), where the arguments are not present but are implied, the complete parentheses must appear even though they are empty.

The general rule that any expression within parentheses is always evaluated first applies also when dealing with functions. Therefore, a function may be given as the argument to another function, as in the following example:

```
X = INT (SQRT (156))
```

All internal functions must appear on the right-hand side of the assignment statement.

For detailed information on how to use each of the functions listed on the facing page, see the chapter "R/BASIC Command Reference."

R/BASIC Internal Functions

@ (at sign)	INT
[] (bracket)	LEN
ABS	LN
ALPHA	MATUNPARSE
ATAN	MOD
BITAND - OR - XOR - NOT	NEG
CALCULATE	NOT
CHAR	NUM
COL1()	OCONV
COL2()	PWR
COS	QUOTE
COUNT	REM

(continued)

DATE()	REPLACE
DELETE	RND (continued)
DIRLIST()	SEQ
DRIVE()	SERIAL
EXP	SIN
EXTRACT	SPACE
FIELD	SQRT
FIELDSTORE	STATUS()
FMT	STR
HASH	SUM
ICONV	TAN
INDEX	TIME()
INMAT()	TIMEDATE()
INP	TRIM
INSERT	XLATE

Correct Use of Internal Functions

X = SQRT(144)

X will be set to the square root of 144.

X = LEN(VAR1)

X will be set to the number of characters in the variable VAR1.

X = DATE()

X will be set to the internal computer date.

External Functions

R/BASIC provides certain commonly used internal functions, but additional functions may be created by the user. These are termed external functions.

Functions

A function is part of an expression and returns one value. The value is used in the expression. It is returned by the RETURN *var* or RETURN *expression* statement. Use an external function in the same manner as an internal function.

External Functions

After an external function has been generated, named, compiled, and cataloged, it can be used by inserting the DECLARE statement at the top of the program and thereafter inserting the function's name into the program as needed.

For example, if an application frequently uses a factorial routine, a programmer might create this factorial function and name it FACTORIAL.

```
FUNCTION FACTORIAL(X)
DECLARE FUNCTION FACTORIAL
IF X EQ 1 THEN RETURN 1
Y=X*FACTORIAL(X-1)
RETURN Y
END
```

The external function named FACTORIAL could be used in a program as follows:

```
DECLARE FUNCTION FACTORIAL
INPUT Y
PRINT FACTORIAL(Y)
```

☛ RETURN *var* must only be used with functions. Subroutines must use RETURN.

For more information, see DECLARE and RETURN in the chapter “R/BASIC Command Reference.”

Correct Use of External Function

```
DECLARE FUNCTION COMP
INPUT INT.RATE
INPUT PERIODS
INPUT SUBTOT
TOTAL.COST = SUBTOT + (SUBTOT * COMP(INT.RATE,PERIODS))
PRINT TOTAL.COST
END
```

```
FUNCTION COMP(INT.RATE, PERIODS)
RATE = (1 + (INT.RATE/100)**PERIODS)
RETURN RATE
END
```

The program uses the declared function COMPOUND to calculate the compound interest and add it to SUBTOT. The result, TOTAL.COST, is printed.

Accessing Advanced Revelation Files

A group of 15 record-oriented commands in R/BASIC allow you to access records in Advanced Revelation or bonded files. Using these commands, you can open files, read and write records or fields, lock and unlock files and records, and perform related file functions.

When executing record-oriented commands, you need not concern yourself with differences in the structure of the actual underlying data. For example, you can use the same READ command to access data in Advanced Revelation, ASCII, and dBASE format (if the

latter two are being accessed with an Environmental Bond). This frees you from having to accommodate the complexities of decoding filing structures or interacting with the operating system. You can use record-oriented commands with any file that can be attached using the Advanced Revelation ATTACH command.

Record-oriented commands can also access a special form of file called a *cursor*. For more information on cursors, see “Cursors” later in this chapter.

Opening a File

Before accessing a file, you must first open it. The command OPEN prepares a file for access, returning a file variable. The file variable is used for subsequent access to the file. You do not need to specify whether you will be reading or writing data — a single form of the command is used to return a file handle for all access. An example program fragment illustrates the use of OPEN:

```
OPEN "CUSTOMER" TO FILE_CUSTOMER ELSE
  MSG("Unable to access %1% file!", "", "", "CUSTOMER")
  STOP
END
```

In this example, the file variable FILE_CUSTOMER is initialized by the OPEN command. All subsequent access to the file will use this variable.

Unlike many programming languages, R/BASIC does not require that you close a file when you are through. This task is handled automatically for you. There is therefore no “close file” command for Advanced Revelation files.

Updating Files

Once the file is opened, you can read, write, or delete data. You can either read and write an entire record at once, or you can access individual fields within a record.

As a rule, a read or write command assumes that a record will be a dynamic array, regardless of the format of the underlying file. Fields in the record are delimited with field marks (ASCII character 254), and further divisions in the record are delimited with value marks, subvalue marks, etc.

Under some circumstances, you may prefer to work with dimensioned arrays. If so, you can use the commands MATREAD and MATWRITE to create and update files. For more information about matrices and arrays, see “Matrices” elsewhere in this chapter.

Record Keys and Select Lists (Cursors)

Record-oriented commands require a record key in order to find the proper record to read, write, or delete. An explicit record key must be included in the read or write command, either as a literal value or in a variable. As an example, here is a small

program fragment that reads one record from a file. The record key is passed as a literal value ("100"):

```

READ @RECORD FROM FILE_CUSTOMER, "100" THEN
  GOSUB PROCESS
END ELSE
  CALL FSMSG()
END

```

If you wish to access a series of records in a file, you can generate a select list (record key list).

Select lists are generated using the SELECT command. There are three variants of the command. The first and simplest version (SELECT) simply provides all the record keys in a file. This is useful if you want to read each record in a file. An example of this version of SELECT might look like this:

```

SELECT FILE_CUSTOMER

```

A second version of SELECT (SELECT BY) allows you to include sort criteria, so that you can get a key list ordered by any field in the record. This form of SELECT returns your select list into one of nine cursors.

An example:

```

SELECT "CUSTOMER" BY "ZIP" SETTING CURSOR_NO ELSE
  CALL FSMSG()
END

```

You can also use this version of SELECT in conjunction with a system subroutine in order to select records out of the file before returning a key list. An example:

```

SORT_LIST = "ST"
REDUCE_SCRIPT = "WITH {ST} EQ 'WA' OR WITH {ST} EQ 'CA'"
REDUCE(REDUCE_SCRIPT, SORT_LIST, MODE, FILE_NAME, ⇐
  CURSORVAR, FLAG)
IF FLAG ELSE
  MSG("Failure in REDUCE","", "", "")
  STOP
END
SELECT FILE_NAME BY SORT_LIST USING CURSORVAR ELSE
  FSMSG()
  STOP
END

```

☛ For more information on the use of REDUCE, see the chapter "R/BASIC System Subroutines."

Finally, you can use the TCL SELECT command. To do so, create a TCL-level SELECT command and use the R/BASIC command PERFORM to execute it from within your program. This version of SELECT allows you to use the full range of SELECT options available at TCL.

Here is an example:

```
COMMAND = "SELECT CUSTOMER BY CITY"
PERFORM COMMAND
```

There is a special R/BASIC command (READNEXT) that fetches the next record key from a select list and makes it available to a read or write process. An extended version of the command enables you to move through the list of keys either forward or backward.

In the following program fragment, READNEXT is used in a loop to fetch each record key from a file. The key is stored in the variable @ID, which is then used in a READ statement to read the record.

```
SELECT FILE_CUSTOMER
DONE = 0
LOOP
  READNEXT @ID ELSE DONE = 1
UNTIL DONE DO
  READ @RECORD FROM FILE_CUSTOMER, @ID THEN
    GOSUB PROCESS
  END
REPEAT
```

Locking and Unlocking

If you are working in a multiuser or multitasking environment, you should make sure that you are coordinating file access with other users. Even if you are in a single-user environment, you should be aware of multiuser and multitasking issues in case your application is moved to a network or a different operating system.

File and record locking is vitally important in applications that will be used by more than one user or session concurrently. Failure to use record locks can result in logical or physical data corruption.

You can assure multiuser coordination by using file and record locking. When you lock a file or record, another user or multiuser session cannot also lock it at the same time. Once you have locked a file or record, you can safely update it. When you are through, you can unlock the record and allow other users access to the data.

A file or record lock does not prevent another user from reading or writing the data you have locked, only from also locking it. See the *Networking* section for more information on using file and record locks.

The LOCK statement sets locks for either files or records. The LOCK statement supports both a THEN and ELSE branch so that you have control over what happens if the lock is successful or fails.

For example, in the following code fragment, the user is prompted to retry if the lock fails:

```

LOCKED = 0; DONE = 0
LOOP
  LOCK FILE_CUSTOMER, @ID THEN
    LOCKED = 1
  END ELSE
    RESP = "R"
    MSG("%1% locked.  R to retry, Q to quit", "R", RESP, "")
  END
  IF RESP EQ "Q" THEN DONE = 1
UNTIL LOCKED OR DONE REPEAT
IF DONE THEN STOP

```

Locks set with the LOCK statement must be explicitly unlocked with the UNLOCK statement. Locks are *not* cleared when a program terminates normally. An option with UNLOCK is the keyword ALL, which unlocks all locks set previously by your workstation.

In R/BASIC you can attempt to lock the same record more than one time. This might happen, for instance, if you run two separate programs at two levels of recursion.

The LOCK statement can tell you when it encounters a locked record whether you have the record lock, or whether someone else does. The value of the STATUS() function after a failed lock is set with these values:

```

0      Locked by someone else
1      Locked earlier by you

```

Here is an program fragment that illustrates how you might check in your program to see if you have previously locked the record:

```

LOCK FILE_CUST, @ID THEN
  GOSUB PROCESS
END ELSE
  IF STATUS() = 1 THEN
    CALL MSG(@ID:" locked by you", "", "", "")
  END ELSE
    CALL MSG(@ID:" not available", "", "", "")
  END
  (etc.)

```

Example Program

The following example program illustrates the general techniques of accessing records in Advanced Revelation or bonded files. The sample program opens a file and establishes a list of record keys. A loop construct is then used to fetch each

record key, lock the record, read it, and then write it back to file. For more information, see LOCK in the "R/BASIC Command Reference."

```

DECLARE SUBROUTINE MSG, FSMSG

* open file. If file not available, quit.
OPEN "CUSTOMER" TO FILE_CUSTOMER ELSE
  MSG("Unable to access %1% file!", "", "", "CUSTOMER")
  STOP
END

* establish "select list" of all records keys
SELECT FILE_CUSTOMER

* use loop to fetch individual record keys, then read and
* write data
DONE = 0
LOOP
  /* fetch next record key. If no more keys are available,
  set flag to exit loop. */
  READNEXT @ID ELSE DONE = 1
UNTIL DONE DO
  * lock record. If record can't be locked, skip update.
  LOCK FILE_CUSTOMER, @ID THEN
    * read record based on key returned by READNEXT
    READ @RECORD FROM FILE_CUSTOMER, @ID THEN
      /* update record (replace first field with
      current date) */
      @RECORD<1> = DATE()

      * write updated record back to file
      WRITE @RECORD ON FILE_CUSTOMER, @ID ELSE
        * if error in writing, post message
        FSMSG()
      END ; * write
      * unlock record
      UNLOCK FILE_CUSTOMER, @ID ELSE
        /* if the record cannot be unlocked, call
        a system error handler */
        FSMSG()
      END ; * unlock

    END ELSE
      * error condition for READ
      MSG("Error reading record %1%", "", "", @ID)
    END ; * read

  END ; * lock
REPEAT
STOP

```


Error Conditions

A variety of conditions can cause a file access command to fail. As a simple example, your program might attempt to read a record that does not exist. A command might also fail because of security violations (no access allowed), a mismatch in data types (attempting to write invalid data to a field), or file corruption.

If this happens, R/BASIC provides you with a means to know the cause and nature of the error. All record-oriented commands except XLATE include an ELSE branch. If the command cannot be completed, the program will branch to the statements following the ELSE branch.

Determining Severity The STATUS() function indicates the severity of the error. Possible values for STATUS() are:

STATUS()	
Value	Meaning
0	Logical error. Examples include a record that does not exist (for READ or DELETE), security violation (insufficient rights to a file), or invalid data type (data does not match the dictionary data type).
1	Physical error. Examples include media full, or access denied by the operating system. You may be able to resolve the problem and retry access.
2	Fatal error. Examples include media error or internal file error. You should not retry access.

Errors of severity levels 1 or 2 are not normally passed back to your program. Instead, these are passed to a standard system error handling routine. For example, a media error is returned to the system error handler, which posts a message and prompts the user to abort the program.

If you wish to trap errors of this severity level, you can set the system variable @FILE.ERROR.MODE. If this variable has a value other than zero or null, the error condition is not trapped by the system. Instead, your program will be responsible for resolving the problem.

@FILE.ERROR.MODE is reset to false (zero) after each file operation. You must therefore reset it to true just before each command if you wish to trap these errors.

Error Detail Details about the error are returned in the system variable @FILE.ERROR. The variable is constructed as follows:

Field	Contents
1	Error code. This identifies the actual error that was encountered. The records FSERRORS_ <i>nnn</i> in the INCLUDE file (where <i>nnn</i> is a century number such as 100) provide descriptions of the error codes used in Advanced Revelation.
2	Error parameters. If the error is accompanied by detail data, this is passed in the second field of @FILE.ERROR. The values in this field are delimited by value marks (@VM).
3	Filing system detail data. If the filing system is returning a non-standard error code, detail information about the error appears in field 3 of @FILE.ERROR.

@FILE.ERROR is a global variable, and its value is reset each time a file operation occurs. This includes not only the next R/BASIC command that accesses a file, but also any CALL, EXECUTE, and PERFORM commands, as well as any calls to external functions or system subroutines. In general, you should read and (if necessary) save the value of @FILE.ERROR as soon as possible after a file operation.

**Handling Failure
Conditions**

If you detect an error in a file operation, you can take two routes. First, you can examine the error by testing @FILE.ERROR, and taking appropriate actions based on its values.

Most filing system errors have an associated message in the MESSAGES file. To call this message, concatenate the literal string "FS" onto the front of the code (for example, code 105 becomes FS105) and call the message using the system subroutine MSG.

The following program fragment illustrates how you might test whether a DELETE operation failed because the record did not exist:

```
$INSERT INCLUDE, FSERRORS_100
DELETE FILE_CUSTOMER, @ID ELSE
  ERROR = @FILE.ERROR<1>
  IF ERROR = FS_REC_DNE$ THEN
    MSG("%1% does not exist!", "", "", @ID)
  END ELSE
    PARMS = @FILE.ERROR<2>
    MSG("FS":ERROR, "", "", PARMS)
  END
END
```

The second method of handling errors is to call the system subroutine FSMSG. This routine is a more convenient method of displaying a message based on the value of @FILE.ERROR. You have the option of adding your own text to this display.

The following program fragment uses FSMSG as a default error handling routine. In this case, all error information arising out of a WRITE statement is passed to the FSMSG routine for display.

```
WRITE @RECORD ON FILE_CUSTOMER, @ID ELSE
  * if error in writing, post message
  FSMSG()
END ; * write
```

Accessing Operating System Files

In addition to the commands that allow you to access Advanced Revelation records, a separate group of commands provides direct access to operating system files (for example, any file that appears in a DOS or OS/2 directory listing). These commands either read and write an entire file at once, or read and write blocks of bytes at a time.

You will use operating system file access commands much less frequently than record-oriented commands. As a rule, operating system access commands are necessary only if you wish to manipulate filing structures directly. This might be required if you were writing a custom import or export process, or creating a new Environmental Bond.

File Commands

The commands OSREAD, OSWRITE, and OSDELETE write an entire operating system file into a variable. For example, you can read the entire AUTOEXEC.BAT file from your system in a single OSREAD command, and write it back out with an OSWRITE command.

Any file that you access using these commands must be less than 64K long. You can use the R/BASIC function DIR to return information about the size of a file before you attempt to read it.

Block Commands

You can access and update files longer than 64K using the commands OSBREAD and OSBWRITE. These commands read or write blocks (up to 64K) of information in operating system files.

Before using OSBREAD and OSBWRITE, you must open the operating system file using the command OSOPEN. As with the OPEN command, this returns a file variable used for subsequent access to the file.

When you use OSOPEN to open an operating system file, you must close the file later with OSCLOSE. This is in contrast to the record-oriented OPEN command, which requires no corresponding close command. If you do not close a file with OSCLOSE, it may not be updated properly, and you may run out of file handles.

The commands OSBREAD and OSBWRITE require that you indicate the starting point and size of the data that you wish to read or write. For example, in the following code fragment, the OSBREAD statement reads 1,024 bytes from a file, starting at byte 4,000:

```
OSBREAD BLOCK FROM OS_FILE AT 4000 LENGTH 1024
```

Error Conditions

R/BASIC uses the function STATUS() to return the success or failure of an operating system file access command. These commands do not use the variables @FILE.ERROR and @FILE.ERROR.MODE.

Because these commands interact directly with the operating system, there is not an intermediate mechanism to trap severe errors such as media failure. These types of errors use standard operating system error messages such as "Abort, Retry, Ignore?".

Possible return values for STATUS() are:

STATUS() Value	Meaning
0	no error
1	bad operating system file name
2	access denied by the operating system
3	disk or directory full
4	file does not exist
5	undefined error
6	attempt to write a read-only file
7	invalid beginning byte position

Example Program

The following program illustrates the use of several operating system file access commands. The program reads information from one file, strips control characters (non-printing characters) from the data, and writes the changed data to a new file.

```
* program to strip control characters from a file
CONTROL_CHAR = ""
FOR CTR = 1 TO 31
  * preserve characters 10, 13 (carriage return, linefeed)
  IF CTR NE 10 AND CTR NE 13 THEN
    CONTROL_CHAR := CHAR(CTR)
  END
NEXT

IN_OFFSET = 0
OUT_OFFSET = 0

* open source file
OSOPEN "WP.DAT" TO IN_FILE ELSE
  CALL MSG("Can't open WP.DAT", "", "", "")
  STOP
END

* create (overwrite) target file and open it
OSWRITE "" TO "WP.OUT"
IF STATUS() THEN
  CALL MSG("Can't initialize WP.OUT file!", "", "", "")
  STOP
END
OSOPEN "WP.OUT" TO OUT_FILE ELSE
  CALL MSG("Can't open WP.OUT", "", "", "")
  STOP
END

DONE = 0
BLOCK_SIZE = 1024
LOOP
  * read 1K from source file
  OSBREAD BLOCK FROM IN_FILE AT IN_OFFSET LENGTH BLOCK_SIZE
  * check to see if the file has been traversed
  IF LEN(BLOCK) LT BLOCK_SIZE THEN
    DONE = 1
  END ELSE
    IN_OFFSET += BLOCK_SIZE
  END

  * convert control characters to null
  CONVERT CONTROL_CHAR TO "" IN BLOCK

  * write (smaller) block to target file
  OSBWRITE BLOCK TO OUT_FILE AT OUT_OFFSET
  IF STATUS() THEN
    CALL MSG("Error %1% writing to file!", "", "", STATUS())
  END
END
```

```

        OSCLOSE OUT_FILE
        STOP
    END
    OUT_OFFSET += LEN( BLOCK )

    UNTIL DONE REPEAT
    OSCLOSE OUT_FILE
    IF STATUS() THEN
        CALL MSG("Error %1% closing file!", "", "", STATUS())
    END
    STOP

```

Cursors

In Advanced Revelation, you can generate a list of record keys (a select list) for use by another process. For example, you can use the TCL command `SELECT` to generate a select list of keys for records that meet specific criteria and are sorted in a particular way. This list can then be passed to the next process, such as `RECORCOPY`, `LIST`, or to your own program.

R/BASIC supports the maintenance of up to nine select lists at once. Select lists are maintained in cursors. At any given `EXECUTE` level, a program has available the default cursor, plus eight global cursors.

A cursor can be thought of and used much like a logical file. Using information stored in the cursor, you can fetch the next record key, read a record, and write a record. Options available in most record-oriented file access commands (`SELECT`, `READ`, `WRITE`, etc.) allow you to specify a cursor in place of a file.

Using cursors for record access also allows you to take advantage of domain-level validation (pattern checking) and conversion. Using this feature, you can have Advanced Revelation verify the correctness of data that is being written to a file, saving you the trouble of coding this yourself. At the same time, Advanced Revelation will convert data from internal format to external format during the read process, and back into internal format during a write operation.

Cursor Attributes

In addition to the list of keys, a cursor is comprised of additional information that R/BASIC uses when accessing the cursor. The specific list of attributes stored with a cursor are:

- The key list
- The file from which the keys were drawn
- The dictionary associated with the data file
- Your current position within the key list

- The selection (reduction) criteria used to generate the key list
- The sort criteria used to generate the key list
- Information about whether the list is complete, or is still being built

The Default Cursor

The default cursor is used if no other cursor is explicitly activated in a program. In essence, the default cursor stores the single active select list available at TCL. All statements making use of a select list in R/BASIC (SELECT, READNEXT, and CLEARSELECT) will use the default cursor if the extended version of these statements is not used.

A default cursor is maintained at each EXECUTE level. If a program moves to new a EXECUTE level (using the EXECUTE statement or the [F5] key), a new default cursor is initialized for that level. The default cursor for the original EXECUTE level is preserved. When the second EXECUTE level is closed (when the program called with EXECUTE stops), the original default cursor again becomes the active one.

Initializing a Cursor

You can have R/BASIC initialize a cursor by using one of three SELECT commands:

- The TCL SELECT command, executed either before you enter your R/BASIC program, or using a PERFORM statement from within your program.
- The basic form of the R/BASIC SELECT statement.
- The extended R/BASIC SELECT ... BY syntax.

Only the extended R/BASIC SELECT ... BY syntax allows you to access more than one cursor. The first two options use only the default cursor.

If you use the extended SELECT ... BY syntax, R/BASIC will return to you a cursor variable (which is very similar to a file variable). You can then use this cursor variable to fetch record keys from the cursor, and to read and write records to a file.

Sorting a Cursor

The R/BASIC SELECT ... BY statement can include an option to return a list of record keys sorted by one or more fields.

Selecting (Reducing) Records for a Cursor

In addition to specifying sort criteria using the SELECT statement, a system subroutine called REDUCE permits the establishment of selection (reduction) criteria from within a program.

- ▮ The REDUCE subroutine is documented in detail in the chapter "R/BASIC System Subroutines."

Accessing a Cursor

A program accesses cursors using the READNEXT statement to fetch record keys, and the normal READ and WRITE statements to update a file using a cursor. If you do not specify a cursor variable when using these commands, the default cursor is assumed.

Each time you execute a READNEXT statement, R/BASIC will return one record key to your program. Your position within the cursor is also updated, so that you can fetch the next key with a subsequent READNEXT statement.

You can also use a cursor to update a file. Most record-oriented commands (READ, WRITE, etc.) allow you to substitute a cursor variable for a file variable. You can therefore fetch a record key using a specific cursor, and then update the file using the same cursor.

Domain-level Validation and Conversion

If you use cursor to access a file, you can take advantage of domain-level validation and conversion. This can save you a great deal of programming, while providing a high level of data integrity.

When you use domain-level processing, all data that you write to a file is validated against the pattern matches stored on the dictionary. Every field in the record being written is checked against the dictionary. If the data is invalid — for example, if some data fails a date conversion — the write operation fails.

Conversely, data that is read from file is passed through any conversions stored in the dictionary before being returned to your program. This means that your program can expect to see data only in external format, and need not include extra logic to convert data from its internal format. If you write the data back to the file, it is converted to internal format for storage.

Domain-level validation and conversion is automatically invoked whenever two criteria are met:

- You are using cursor access to a file.
 - The file being accessed has had control features added to it. You can add control features to a file by using the CONTROL_ON command at TCL.
- ☞ For more information on control features for a file, see the chapter “Managing Files and Records” in the section *Filing Systems*. For information on adding and removing control features, see the TCL commands CONTROL_ON and CONTROL_OFF in the “TCL Command Reference” chapter.

Bidirectional Cursors

In addition to being able to access more than one cursor, a program can indicate the direction in which to read the cursor.

The READNEXT statement in R/BASIC includes options for ascending and descending cursor access. Ascending access to a cursor will present record keys starting

from the top of the list to the bottom. Descending access presents keys in the opposite manner, providing those at the bottom of the cursor first.

A program can change its READNEXT direction dynamically. This permits a program to “browse” through a list in both directions.

- ☛ The words “ascending” and “descending” in this context do not necessarily imply a sorted order to the record keys. As used here, the two words simply indicate a direction.

Terminating and Nonterminating Cursors

Any cursor can be treated as terminating or nonterminating. READNEXT can specify whether a cursor is to be terminated, or cleared, when it has been completely read. If a cursor is to be terminated, the entire list is cleared after the last record key has been returned by READNEXT.

In nonterminating mode, READNEXT does not clear the list when done. Instead, when the last key in the cursor has been read, READNEXT returns to the beginning of the list and continues reading. The cursor is thus circular; there is no logical end to the list.

Resolved vs. Latent Key Lists

Record key lists in cursors can be either “resolved” or “latent.” Under most circumstances, it is not critical that a program detect the type of list that it is processing. In fact, the difference between the two types of lists is transparent to a program unless you explicitly test for the difference.

However, it is sometimes useful or necessary to know whether an active list is latent. This might be the case if the program is updating the file, or if you wish to avoid executing extra READ statements.

- ☛ For information on the difference between resolved and latent key lists, see the “Introduction to R/LIST” chapter in *R/LIST*.

If necessary, a program can test for a latent select list using the system variables @LIST.ACTIVE and @REDUCTION.DONE. These are documented under “Special R/BASIC Variables.”

A program that updates the file from which it is drawing keys might behave unpredictably if using a latent list. Because a latent list is being created dynamically, the list of records to be processed is not known. A program that updates the file being processed might create records that in turn are processed as the program gets to them.

For example, using a latent select list, a program might read a record, and based on data in the record, create a new record in the same file. The program writes out the new record under a different key. It is possible that as the program proceeds along the file it will encounter and process this new record.

This might not always be an undesirable result. Nonetheless, you should be aware of this possibility. If a program will update a file, it is likely that a resolved select list is the better basis for processing.

- ☛ To guarantee that a select list is not latent (in other words, is resolved), use the syntax `PERFORM "SELECT file ..."` in an R/BASIC program.

Avoiding Extra Reads

A program processing a latent list can avoid executing an extra `READ` statement, one for selecting and one for processing. This results in faster processing.

Latent lists are resolved directly in the `READNEXT` statement. In other words, `READNEXT` reads a record, and then determines whether the record belongs in the cursor. Because `READNEXT` has already read the record, no extra read is required.

You can use the system variable `@REDUCTION.DONE` to determine whether `READNEXT` will read the record. For more information on the use of this system variable, see "Special R/BASIC Variables" later in this chapter.

Cursors and File Assignments

All cursors are associated with a specific file. When a cursor is initialized with a select list, it is tagged with the file from which the keys were drawn.

Occasionally you will want to generate a cursor using one file, and apply it to another file. For example, a program might use a `SELECT` command to generate a list of keys from one file, and then use this list to delete records from a second file.

In order to achieve this, the cursor must be assigned to a new file. Special syntax is available in the R/BASIC `SELECT` command to make this transfer of file names from one cursor to another.

The R/BASIC Debugger

The Debugger allows you to break into a program during its execution. From within the program, you can display the content of its variables; change the content of its variables; set break points, and trace variables, program names, and line numbers.

Activating the Debugger

Activate the Debugger in any of the following ways.

1. Press the [Ctrl-Break] key. (The Break key may be inhibited by the `BREAK KEY` statement. If so, you cannot access the Debugger. See `BREAK` in the chapter "R/BASIC Command Reference.")

2. Use the DEBUG statement anywhere in a program where a statement can be used. It stands on a line by itself. The format is:

```
DEBUG
```

3. Use the D option with the RUN command. The format is:

```
RUN filename program.name (D)
```

When the program is under control of the Debugger, it displays the following message:

```
Line n 'file program' broke because the break key was hit.
```

The Debugger uses the exclamation point (!) as a prompt. Type HELP to display a message containing a summary of the debugging commands.

The Debugger Command Summary following lists the commands in categories by their function. Each command is described in more detail in an alphabetical listing that follows.

Programs must be compiled with LINEMARKS or contain explicit LINEMARK statements for effective use of the Debugger. See LINEMARK in the chapter "R/BASIC Command Reference."

Table N3-1

Debugger Commands

Topic	Command	Summary
Program Control	[Esc] or END	Aborts program and returns to TCL
	EXECUTE <i>command</i>	Executes a command as though at TCL
	G	Go. Resumes program control
	OFF	Return to DOS
Source Code	\$ or ?	Displays the current line and program name

Table N3-1

Debugger Commands (*Continued*)

Topic	Command	Summary
	<i>En</i>	Sets multistep line counter to <i>n</i>
	<i>Gn</i>	Goes to line <i>n</i>
	<i>Ln</i>	Lists <i>n</i> lines from current line
	SO{URCE}	Specifies the filename and program of the source code
	ST	Toggles the Source Trace mode on or off
Variable Content	/x	Displays the content of variable <i>x</i> and prompts for any new value
	\x	Displays the content of <i>x</i> in hexadecimal code
Break Table	B@= <i>program</i>	Breaks if program name = <i>program</i>
	B&= <i>n</i>	Breaks if a line number equals <i>n</i>
	Bx operator constant	Breaks when a variable, <i>x</i> , meets certain conditions
	K	Clears the break table
	<i>Kn</i>	Clears the break table entry <i>n</i>
Trace Tables	D	Displays the break table
	T	Toggles the trace mode on or off
	T\$	Traces the source of a program
	T@	Traces a program source
	T <i>x</i>	Traces a variable, where <i>x</i>
Program Stack	U	Clears the trace table
	<i>Un</i>	Clears the trace table entry
	D	Displays the trace tables
	PS	Displays the program stack

\$, ?

Type either \$ or ? to display the current program's name and the most recent line of executed code.

Bx operator constant

A program can be broken when the contents of a variable have a given relationship to a constant. For instance, the command `BTOTAL > 500` says to break if variable `TOTAL` is greater than 500. The operator can be `=`, `#`, `<`, `>`, `<=`, `>=`, or `=`. The constant specifies the contents of the variable.

B\$

The \$ character indicates the current program line number. For example, `B$ = 3` causes a break when the current program reaches line 3.

B@

The @ character indicates the current program name. For example, `B@ = EXAMPLE` causes a break when the current program name is `EXAMPLE`. This feature is useful when you deal with many subroutines.

D

The D command displays the contents of the break and trace tables. For example, when the following commands are entered at the Debugger:

```
!B@=SUBROUTINE1 +
!B$=24 +
!BTOTAL 500 +
!T@ +
!T$ +
!TTOTAL
!D
```

the trace table will display:

```
1 break in program 'SUBROUTINE1' if (PROGRAM) = SUBROUTINE1
2 break in program 'BP EXAMPLE' if (LINE) = 24
3 break in program 'BP EXAMPLE' if TOTAL GT 500
1 Trace program name once upon entry.
2 Trace every line in every program.
3 Trace variable TOTAL in routine BP TMP.
```

E

Clears the multistep line counter.

En

You can single-step through the program by typing an E1 and then a G. The number following the E indicates the number of lines of R/BASIC code that should be executed before returning to the Debugger. If no number is specified, then line counting is disabled and execution may continue normally.

[Esc] or END

Press [Esc] or type END to terminate the current program.

EXECUTE

Use the EXECUTE command to recursively execute any TCL command from the Debugger. Control will return to the Debugger. For example, the command:

```
EXECUTE EDIT BP EXAMPLE
```

will allow the user to edit a program in the BP file called EXAMPLE. When the edit is completed, control will return to the Debugger.

- ☛ When using the networking version of Advanced Revelation, entering [Esc] or END will unlock all records that have been locked by the program.

G

Type G to transfer control from the Debugger and continue the program execution. The screen will redisplay its original contents.

Gn

To go to a specified line in the source code, type Gn where n is the line number in the source program.

K, Kn

The break table can hold any number of entries. One entry is written for each use of a B command. It is cleared by using the K command. A specified BREAK command can be cleared using Kn where n is the break table number of the command.

Ln

Display source code by typing Ln where n is the number of lines to display, starting at the present line.

OFF

Type OFF to log off Advanced Revelation and return to the operating system.

PS

The program stack is an array of the programs currently residing in memory. The most recent programs are added to the end of the stack. The PS command will display the contents of the program stack.

For more information, see BREAK, DEBUG, EXECUTE, LOCK, and UNLOCK in the chapter "R/BASIC Command Reference," and the RUN and OFF commands in *TCL*.

SO{URCE}

SOURCE and its abbreviation SO can be used to link the source code of another program. By specifying a file and program name, you can view any program in an attached file.

ST

Type ST to toggle source code trace mode on and off. Source trace mode will print the line of source code that is being executed.

T

The T command toggles a trace display flag. It temporarily turns the trace table on or off.

Tx

The contents of a variable will be displayed at every line if a T (trace) followed by the variable name is entered. For example, TTOTAL would trace variable TOTAL. The Debugger keeps a table of the variables to be displayed at every line. Any number of variables may be added to the trace table in this manner.

T\$

Enter a \$ character after the Trace command to print the current line number at each line.

T@

Use the @ character to print the program name each time a new program is entered.

U

Clears the Trace table completely.

Un

Clears a single entry in the Trace table. The n specifies the number of the entry to clear.

/x

To see the contents of a variable, type a slash mark (/) followed by the variable name. The Debugger prints the contents of the variable followed by an equal sign (=). To change the contents of that variable, enter the new data after the equal sign (=).

To see the contents of system variables, type the variable name after the slash mark (/).

To return to the Debugger exclamation point prompt without changing the variable, press the [Enter] key.

To force a null value into a variable, type two single quote or two double quote marks. If you press [Esc] or type END, the variable is left unchanged. (The END can be forced into a variable by surrounding it in double or single quote marks.)

\x

Display the contents of a variable in hexadecimal format using the back slash mark (\).

Interfacing Dictionaries and R/BASIC

Complex dictionary words can be created by using any and all the components of R/BASIC. An R/BASIC program can call a dictionary word by using the {} (braces) function.

This section deals with the interfacing of R/BASIC and dictionaries. You should be familiar with both R/BASIC and dictionary concepts.

This topic has been broken into two parts. The first covers the use of dictionary words in an R/BASIC program using the {} (braces) function. The second discusses the technique of using R/BASIC structures to create dictionary words.

The {} (Braces) Function

Sometimes it is desirable to refer to a dictionary field name in an R/BASIC program. Several special R/BASIC variables are used to interface R/BASIC to the dictionary word:

@DICT	file variable for the dictionary of the primary file
@ID	record key of the current record
@RECORD	current record

The @ character indicates to the R/BASIC compiler that a special variable name follows.

For instance, assume that you want to use the dictionary word {INV.AMT} from the INVOICE file. Also assume that the invoice file contains 10 records numbered 11 to 20. The following code would print the INV.AMT for each of the 10 invoices:

```

OPEN 'INVOICE' TO FILE.IV ELSE
  CALL MSG("No INVOICE file!", "", "", "")
  STOP
END
OPEN 'DICT.INVOICE' TO @DICT ELSE STOP
FOR @ID = 11 TO 20
  READ @RECORD FROM FILE.IV, @ID ELSE @RECORD = ''
  PRINT @ID "L#3" : {INV.AMT} "R#10"
NEXT @ID
END

```

Notice that the dictionary of the INVOICE file is opened to @DICT. If the dictionary were not opened to @DICT, the word {INV.AMT} would not have been loadable and a runtime message would have occurred.

This error can also occur if {INV.AMT} is not a valid dictionary word in the INVOICE dictionary.

Line 5 of the program shows that the special variable @ID is to take on the values from 11 to 20. @ID must always contain the record ID of the record to be processed. It is a good programming practice to use it to read @RECORD as in line 6. If @ID or @RECORD were mis-assigned when the {INV.AMT} function was called, the resulting errors would be unpredictable.

The {INV.AMT} syntax on line 7 is compiled as a function call. (A function is similar to a subroutine except that the function leaves a value in its place upon returning.)

The first time {INV.AMT} is executed, it is loaded from the dictionary of INVOICE. It remains loaded until the program ends.

For more information, see CALCULATE in the chapter "R/BASIC Command Reference."

Creating Dictionary Formulas Using R/BASIC

This section deals with the creation of dictionary formulas using the syntax structures of R/BASIC. The following special variables can be used:

@ID	set by R/LIST and Window to the current record ID
@RECORD	set by R/LIST and Window to the current record
@ANS	calculated result or answer

The R/LIST processor reads the current record into @RECORD by using @ID as the record key. This allows you to manipulate the information contained in the current record to produce complex results. Any and all the structures of R/BASIC may be used. The resulting answer must be stored in @ANS.

The dictionary type must be S (symbolic) and the formula must be stored in field 8 of the dictionary item. Multiple statements can be separated by value marks or by the end-of-statement marker (;). The {} (braces) function may be used as described in the previous section.

For instance, the following dictionary formula calculates {SALES.TAX}:

```
@ANS={INV.GROSS} * 5.4 ; @ANS=INT(@ANS/100)
```

Notice that the answer is stored in @ANS and that it may be used to store a temporary result. Notice also that the word {INV.GROSS} is referenced. It must exist as a valid dictionary record in the same dictionary.

Just as in R/BASIC, multiple statements can occur on a single line if separated by a semicolon (;). Here is an example of the formula for the invoice total:

```
@ANS = {INV.GROSS}+{SALES TAX}
```

The XLATE function can be used to translate from the current file to another file to retrieve a value. See XLATE in the chapter "R/BASIC Command Reference."

```
@ANS = XLATE("CUSTOMERS", {CUST.NO}, 5, "X")
```

This formula will use the customer number from the current file (INVOICE) and retrieve field 5 from the CUSTOMERS file.

Dictionary formulas may be up to 34,000 characters in length.

Special R/BASIC Variables

Special R/BASIC variables allow R/BASIC to interact with various modules of the system, including TCL, R/LIST, and dictionaries. All variables are preceded by an @ (at-sign).

Below is a list of special R/BASIC variables and a short description of each.

Cursors and Select Lists

The following variables are used for maintaining and querying the status of record key lists.

- | | |
|-----------------------------|--|
| @FILTER | The name of the most recently selected Filter (this name is set or reset when a filter is selected and before it is executed). |
| @LAST.SELECT.PROCESS | The name of the last process used to generate a list of record keys (used by the system to link list processes and messages). |

@LIST.ACTIVE	The flag which states whether there is an active select list in cursor 0, and what type of list it is. Possible values for @LIST.ACTIVE are: null No select list active 1 Latent file-based list active 2 Latent index-based list active 3 Resolved list active
@PRI.FILE	The file handle for the file represented in cursor zero.
@QUERY.DICT	The file handle for the dictionary represented in cursor zero.
@RECCOUNT	If there is an active, resolved select list, this variable contains the number of record keys in the list.
@REDUCTION.DONE	Contains 0 (zero) if the record key list in cursor 0 is completely resolved, otherwise it is null.
@RN.COUNTER	The READNEXT counter. Incremented for each record examined by the READNEXT statement. It must be initialized by the program to a starting value (usually zero).
@SAVE.SELECT	If set to 1 (true), this preserves an active select list when returning from a main program to TCL.

Editor

The following are variables that affect the operation of the editor.

@CUR.BUF	Contents of current cut-and-paste buffers. Field one contains the current buffer number; subsequent fields contain the text for specific buffers.
@DEST.ED	Contains the flag set at the Destructive Edit prompt in the Environment window. Set (Y at prompt) if 1.
@INSERT	Toggles for auto-line mode in the editor. Field values are: 1 word wrap on 2 wrap line on 3 insert line on 4 indent line on
@TAB.STOPS	A multivalued list of tab positions for use in windows and the editor.
@TYPEAHEAD	Flag indicating whether keystrokes will be buffered. Established in the Environment window.

Environment

A number system variable allow you to query (and set) the current Advanced Revelation environment.

@ACCOUNT	The name of the current account.
@CPU.TYPE	Returns a value corresponding to the type of CPU on which Advanced Revelation is running. For example, an IBM PC/AT will return the value 286.
@CRTHIGH	Height of CRT screen. Used by R/LIST and some TCL commands.
@CRTMAXHIGH	The maximum number of lines on the monitor.
@CRTMAXWIDE	The maximum number of columns on the monitor.
@CRTWIDE	Width of the CRT screen. Used by R/LIST and some TCL commands.
@ENVIRON.KEYS	Contains the user name of the environment and video settings, respectively, for the current user as established in SecureUser.
@ENVIRON.SET	Array of environmental settings for the current user. For positions, see ENVIRON.CONSTANTS in the INCLUDE file.
@EXIST8087	Value is 1 if an 80x87 math coprocessing chip is installed, and 0 (zero) if the math coprocessing emulator is running.
@FILES	Dynamic array of currently attached files.
@FILES.SYSTEM	Contains a list of files that cannot be detached.
@HELP.LEVEL	The Help level for the current user. Established when a user first logs into Advanced Revelation and set in the Environment window.
@LAST.ERROR	The most recent system error message number.
@LEVEL	The current PERFORM/EXECUTE level. The number is incremented with each PERFORM or EXECUTE and decremented with each TCL-level RETURN.
@LOWER.CASE	A string of lower case alphabetic characters used in converting to lower case.
@LPTRHIGH	Height of the printer page. Used by R/LIST and some TCL commands.

@LPTRWIDE	Width of the printer page. Used by R/LIST and some TCL commands.
@MACRO.KEYS	Array of codes and commands for the currently active macro key set. Field one contains codes, @VM-delimited; field 2 contains commands, @VM-delimited. Field 3 contains the macro set name.
@MESSAGES	Array (@RM-delimited) of message records currently being buffered. Messages are buffered by being tagged as "B" in the MESSAGES file.
@PDISK.ON	If printer output has been redirected to a file using PDISK, this variable is set to true. This variable also serves as a switch for the printer check in the R/BASIC PRINT statement. If the @PDISK_ON is false, the PRINT statement will attempt to verify that a printer is available before proceeding. If @PDISK_ON is set, this test is skipped.
@PRINTMODE	If this variable is set to true, the R/BASIC PRINT function will print the ASCII characters zero through 31 using their assigned graphics. If set to false, these characters are not printed, and are treated as control characters.
@ROLLOUT.FILE	The name of the operating system file that is set by ROLLOUT and used by SUSPEND (MS-DOS version only).
@SENTENCE	The most recent TCL sentence.
@STATION	The network station identifier.
@TCL.STACK	Dynamic array of the last commands typed at TCL.
@UPPER.CASE	A string of upper case alphabetic characters used in converting to upper case.
@USERNAME	Contains the name under which the current user logged in. If no user name was specific at login, contains the current account name.
@VOLUMES	A dynamic array of attached volumes.

File Access

The following variable control and report on errors in file access.

@FILE.ERROR Array of error information returned if a record-oriented file access operation fails. There are three fields; their layout is:

Field	Description
1	a message number associated with the current error (the corresponding message in the MESSAGES file will have "FS" prefixed).
2	parameters to be filled into the text of the message, if any.
3	further detail information if the error is not a standard error.

@FILE.ERROR.MODE If true, causes R/BASIC to pass errors of severity greater than 0 (STATUS() = 1 or 2) during a WRITE, DELETE, and READ back to the program. If false, errors of severity greater than 0 are passed to the system error handler. This variable is reset to false after each READ, WRITE, and DELETE operation.

Files and Dictionaries

Use the following variables when interfacing files and dictionaries with R/BASIC.

- @ANS** The result of a symbolic dictionary word. Also used by some system subroutines (example: CATALYST) to return values.
- @DICT** The dictionary of the primary file.
- @ID** The record key of the current record.
- @MV** The value number of a multivalued field when the {} braces function is used (@MV = 0 means all values)
- @RECORD** The current record.

Index

The following variables are used with indexing.

@BACKGRND.TIME The number of seconds that the background processor waits before checking for index transactions to complete (used after the processor has finished the available transaction).

- @DEFAULT.STOPS** A dynamic array of words to skip when cross referencing (the delimiter is @VM).
- @INDEX.TIME** The number of seconds to wait at a prompt before calling the background index process.

Input Characters

The following variables are used with input character streams.

- @CAPTURE** The characters that are used to operate the system in the CAPTURE PLAY process.
- @DATA** The value set by the most recent DATA statement where each element is separated by a carriage return (ASCII character 13).
- @PLAYBACK** The name of the record that contains the character stream currently used in the CAPTURE PLAY process.
- @PLAYDELAY** The number of delay loops used to slow down the CAPTURE PLAY process.
- @PROG.CHAR** The last character typed.
- @SCRIPT** The characters that are being recorded by the CAPTURE process.

Keystroke Arrays

The following variables contain arrays (delimited with @FM) of keystrokes used by Advanced Revelation. For information on the layout of each array, see the record EDIT.KEYS in the INCLUDE file.

- @EDIT.KEYS** Array of keystrokes used for editing (example: insert toggle key).
- @INT.CONST** Array of keystrokes used globally in windows (example: [Ctrl-F8] to move a window).
- @MOVE.KEYS** Array of keystrokes that move the cursor (example: [PgUp]).
- @PRIORITY.INT** Array of keystrokes active at all times (priority interrupts) (example: macro keys, [F5]).

R/LIST and Queries

The following variables are used to maintain information about the current report.

- @BREAK** The most recent value used by the BREAK-ON process in R/LIST.

@HFACTIVE	Toggle for activating or deactivating the current R/BASIC HEADING, FOOTING, and paging logic. If true, the current heading and footing are in effect, and page logic is on. If false, no heading, footing, or page logic is in effect. Heading and footing text is established using the R/BASIC HEADING and FOOTING commands.						
@PAGE	The current page number set by R/LIST.						
@QUERY.DEPTH	Contains the number of queries that will be saved in the Query table.						
@QUERY.TABLE	Contains a copy of the query table for the current user. Each field points to an entry saved in the LISTS file. There are three values in each field: <table> <tr> <td>1</td><td>query record key in LISTS file</td></tr> <tr> <td>2</td><td>command used to generate query</td></tr> <tr> <td>3</td><td>number of records selected in query</td></tr> </table>	1	query record key in LISTS file	2	command used to generate query	3	number of records selected in query
1	query record key in LISTS file						
2	command used to generate query						
3	number of records selected in query						
@VIEW.MODE	A flag used with the system subroutine INIT.VIEW to initialize and display a View window.						

Status Line

@STATATR	Array of status line definitions. For layout, see STATUS.CONSTANTS in the INCLUDE file.
@STATLIST	Buffer for status list templates defined in the STATUS.LIST record of the SYSTEM file. These templates are loaded into @STATLIST at login.
@STATPOS	Array of video, location, and format parameters for each cell in the status line. Each cell is a field. The first value of each field contains the video and location attributes of the cell. The second value contains the format specification.
@STATREC	Current status line record. Each value is one cell.
@STATUS.ON	The status line flag (if true, the status line is displayed).

System Delimiters

The following variables contain characters that are recognized by Advanced Revelation as data delimiters.

@RM	A record mark (ASCII 255). Separates records.
@FM	A field mark (ASCII 254). Separates fields within a record.
@VM	A value mark (ASCII 253). Separates values within a field.

- @SVM** A subvalue mark (ASCII 252). Separates subvalues within a multivalued field.
- @TM** A text mark (ASCII 251). Separates lines in a text field.
- @STM** A subtext mark (ASCII 250). Separates sublines of text.

User-Defined Variables

The **@USER** (**@USER0**, **@USER1**, ... **@USER4**) and **@RECUR** (**@RECUR0** ... **@RECUR4**) series allow the user to interface between main routines and recursive levels. These variables are set to null at LOGON. Any and all use of these variables is left to the user.

The **@USER** variables are not recursive. That is, they retain their values as set by the user's applications and are not affected by the EXECUTE or PERFORM statements.

The **@RECUR** variables are recursive. That is, they are saved whenever a PERFORM or EXECUTE occurs, and restored when returning to the original level. It is therefore possible to maintain up to nine sets of **@RECUR** variables (one for each possible EXECUTE level).

- @USER0 - @USER4** Open for use by the user.
- @RECUR0 - @RECUR4** Open for use by the user.

Video Attributes

The following variables are arrays of Advanced Revelation-specific video sequences. For specifics on the layout of each "W" array (example: **@PW**), see WINDOW.POINTERS in the INCLUDE file.

- @ATRBT** An array of all video attributes available to the system.
- @ATRBT.PTR** An array of mnemonic names for the video attributes in **@ATRBT**.
- @AW** The application window video attributes.
- @EW** The message window video attributes.
- @MW** The menu video attributes.
- @HW** The Help window video attributes.
- @PW** The Popup video attributes.
- @SW** The subvalue window escape codes.
- @VW** The multivalue window escape codes.

@XW The Zoom window video attributes.

Windows

The following variables are used specifically by the window processor.

@BROWSE.LIST	Contains the current browse list. This variable can be initialized by the programmer.
@BROWSE.MODE	True if a browse list is active.
@MV	The number of the value being processed in a multivalued prompt.
@PRIVILEGE	Window restriction level for current user as established using the SecureUser window.
@PSEUDO	The variable used to pass data between windows. This variable is typically used with collector windows.
@TUTOR	The name of the current window or process within a window.
@WINDOW.LEVEL	Contains the indicator for the number of windows currently displayed (as displayed in the status line). This is a window level only, and does not indicate the current level of recursion. There is no limit (except RAM) to the number of levels to which windows can be displayed.

AREV.COMMON

The window processor maintains a list of 133 COMMON variables known as the AREV.COMMON variables. These variables are used to store information such as general window attributes and specific prompt attributes.

Using AREV.COMMON

If you use an \$INSERT statement or otherwise declare the variables, you can access and change their values. Doing so can affect the way a window runs. For instance, you can access the data input by a user before it is put into @RECORD, change the prompt order of fields on the screen, create or clear browse lists, or even change the actual prompts that appear in a window.

The entire AREV.COMMON block is available in the INCLUDE file, and the variables can be included in your programs with an \$INSERT statement.

AREV.COMMON in Main Programs

The common block in the INCLUDE is suitable only for insertion into subroutines that will be called by the window processor. If you are using AREV.COMMON in a

main program, or in a subroutine that is not being called from a window, a slight change needs to be made to the block.

A good strategy is to copy the AREV.COMMON block (using RECORDCOPY) to the same file under a new name. For instance, create a new record called AREV.COMMON.MAIN with this command:

```
RECORDCOPY INCLUDE AREV.COMMON TO: AREV.COMMON.MAIN
```

Then change the line reading COMMON W(W.CNT). Instead of reading COMMON W(W.CNT), the line should be changed to read COMMON W(1). This will prevent the program from aborting with a variable not assigned a value error at runtime.

AREV.COMMON Examples

In the following pages, the most commonly used AREV.COMMON variables are described. A short description of each variable is given. In many cases, a short program or program fragment illustrates a typical use of the variable.

AMV.ACTION The associated multivalue (AMV) action variable is code indicating the action taken on an AMV group. The codes for AMV.ACTION are:

Code	Meaning
0	No action
1	Clear the AM
2	Delete the AMV
3	Insert a value in the AMV
4	The AMV has been changed

The following code is an example (called as a Perpetual process):

```
IF AMV.ACTION = 2 AND MV = 1 THEN ⇨
    AMV.ACTION = 0
```

This will prevent the deletion of the first value in any AMV group in the window.

BROWSE.NEXT The BROWSE.NEXT variable indicates the next browse list to be processed. Keys are separated by value marks. For example:

```
BROWSE.NEXT = 'KEY1':@VM:'KEY2'
BROWSE.LIST = BROWSE.NEXT
NEW.BROWSE = TRUE
RESET = RESET.RECORD$
```

CURR.AMV.GROUP The number of the current AMV group. If no AMV group is active this variable is null.

DATAFILE The name of the primary datafile being processed by the window. The datafile may be changed, for example, from TCL with a command such as:

```
WINDOW TEMPLATE.NAME DATAFILE = ⇨  
    ALTERNATE.FILE.NAME
```

Note that there are no spaces on either side of the equals sign and no quotation marks around the alternate file name.

DISPLAY.ACTION A code indicating how the system should display the current record. **DISPLAY.ACTION** can be used to force the redisplay of **@RECORD** or the recalculation of symbolic prompts. The code can take one of three values:

DISPLAY.ACTION Value	Meaning
5	Redisplay all prompts
6	Redisplay a list of prompts (REDISPLAY.LIST)
9	Recalculate prompts (RECALC)

For example, as part of a pre-prompt process:

```
IS='NEW DATA'  
DISPLAY.ACTION = REDISPLAY.PROMPT$
```

IS The data in the current field. Can be used in conjunction with **VALID** to do data validation from post-prompt processes. For example, as a part of a post-prompt process:

```
IF IS NE IS.ORIG THEN  
    VALID = FALSE  
    CALL MSG('NO CHANGES ALLOWED',' ',' ',' ')  
END
```

IS.DFLT The default value of the current prompt. **IS.DFLT** can be created or modified from a pre-prompt process but may only be inspected from the Default prompt. (To establish a default value from the Default prompt of the Prompt window your code must set **@ANS**.)

This code will provide a default value if the field is empty. (Called as part of a pre-prompt process.) For example:

```
IF NOT(IS) THEN IS.DFLT = 'NEW DATA'
```

IS.ORIG The original value of the data in the current field. That is, the data before any changes were made.

MV The current value position of the current AMV group.

MV.NEXT The next value position in the current AMV group.

The following section of code will let the user decide which value they want the cursor to be on. This would be called from the [F2] Options key.

Note that RESET is used to move the cursor:

```
REPLY = ''
NUM.VAL = COUNT({Field}, @VM) + 1
CALL MSG('Which value?|⇨
(1 to %1%)', 'R', REPLY, NUM.VAL)
IF NUM(REPLY) AND REPLY NE NUM.VAL THEN
  MV.NEXT = REPLY
  RESET = RESET.PROMPT$
END
```

NEW.BROWSE Flag indicating whether a new browse list is active. Used in conjunction with BROWSE.NEXT to activate a new browse list.

OREC The value of the record as it was read off of disk. Compare @RECORD to OREC to see if any changes have been made. Useful in pre-read, pre-save, and pre-delete processes. For example:

```
IF @RECORD = OREC THEN...
```

PKEY The value of the previous key.

The following code fragment, called as a post-initialize process, establishes PKEY and PREC so defaults of " (double quotes) and [ALT-O] work for the first record processed by the window. The code assumes a single part key with a sequential counter (%SK%).

```
READV PKEY FROM SRC.DICT, '%SK%', 1 ELSE STOP
PKEY = PKEY - 1
READ PREC FROM SRC.FILE, PKEY ELSE NULL
```

PREC The value of the record processed prior to the current record. PREC can be used to determine default values or to preload specific fields for the user.

From a post-read process the following code can be used to preload specific fields:

```
IF OREC='' AND PREC NE '' THEN ⇨
  @RECORD<1>=PREC<1>
```

RESET A code used to tell the system what part of the process to reestablish. The code may take one of six values:

RESET Code	Meaning
1	Clean up line
2	Clean up edit
3	Reset edit
4	Reset prompt
5	Reset record
6	Reset window

For example, use this to exit a window:

```
RESET = RESET.WINDOW$
```

SI A dynamic array containing the current prompt's characteristics. For example:

```
* put 100 in prompt register 1 for current prompt
SI<PREG1> = 100
```

SRC.DICT The file variable of the dictionary for the datafile used by the window.

SRC.FILE The file variable for the data portion of the file used by the window. It would be needed by replace read/save/delete processes.

For example below is the system variable that holds the user's restriction level. It could be called as part of a Replace Write process.

```
IF @PRIVILEGE GT 3 THEN WRITE @RECORD ON ⇐
    SRC.FILE, @ID ELSE
    CALL FSMSG()
END
```

VALID If VALID is set to 0 (FALSE) then the current process is aborted. It can be used when doing edit checks from a post-prompt process or as part of pre-read/save/delete processes.

The following is an example called as part of a pre-delete process:

```
IF @RECORD<1> NE '' THEN VALID = FALSE
```

Example called as part of a post-prompt edit check:

```
IF IS = 100 THEN VALID = FALSE
```

- W()** A dimensioned array containing the prompt characteristics of all prompts on the window.
- W.CNT** The count of the number of prompts on the window. Used to dimension W.
- WCHANGE** A flag indicating whether the data in the current field has been changed. True (1) indicates a change has been made.
- WI** The number of the current prompt. For example,
- W(WI) <PREG1>
- W(WI) is equivalent to:
- SI<PREG1>
- WI.NEXT** The prompt that will be processed next. Normally this is the next prompt in the prompting sequence but it can be altered by the program. If altered from a pre-prompt process, RESET will have to be set. For example, the following illustrates WI.NEXT changed from a pre-prompt process:

```
IF IS THEN
  WI.NEXT = 3
  RESET = RESET.PROMPT$
END
```

This example illustrates WI.NEXT changed from a post-prompt process:

```
IF IS = IS.ORIG THEN WI.NEXT = 5
```



Chapter N 4

R/BASIC Command Reference

This chapter is an alphabetical listing of all the commands and functions in R/BASIC. Each entry contains:

- A complete description of the command.
- The command syntax.
- Details about each argument or parameter.
- Examples showing you how to use and how *not* to use each command.

If you are not sure what command you need, see the chapter “Features of R/BASIC” earlier in this section. For general information on programming in R/BASIC, see the chapters “Introduction to R/BASIC” and “Programming with R/BASIC”.

+*, !, REM, /* ... */ (Comments)	N 4 . 1
;(Multiple Statement Operator)	N 4 . 2
[] (Bracket)	N 4 . 3
{ } (Braces)	N 4 . 6
\$INSERT	N 4 . 7
+++, --, ***, ///, ::: (Special Multivalue Arithmetic Operators)	N 4 . 9
(Angle Bracket Operators)	N 4 . 10
= (Assignment Statement)	N 4 . 11
=, +=, -=, := (Special Assignment Operators)	N 4 . 13
=, +=, -=, := (Assigning Matrix Elements)	N 4 . 14
ABS	N 4 . 18
ALPHA	N 4 . 19
AND and OR	N 4 . 20
ATAN	N 4 . 21
BITAND, BITOR, BITXOR, BITNOT	N 4 . 22
BREAK KEY	N 4 . 24

CALL	N 4 . 26
CASE	N 4 . 27
CHAIN	N 4 . 29
CHAR	N 4 . 30
CLEAR	N 4 . 31
CLEAR COMMON	N 4 . 32
CLEARFILE	N 4 . 33
CLEARSELECT	N 4 . 34
COL1()	N 4 . 35
COL2()	N 4 . 36
COLHEADING	N 4 . 37
COLLENGTH	N 4 . 38
COMMON	N 4 . 39
Labeled COMMON	N 4 . 41
CONVERT	N 4 . 42
COS	N 4 . 43
COUNT	N 4 . 44
DATA	N 4 . 46
DATE()	N 4 . 47
DEBUG	N 4 . 48
DECLARE	N 4 . 49
DELETE	N 4 . 50
DELETE()	N 4 . 51
DIMENSION	N 4 . 53
DIR()	N 4 . 54
DIRLIST()	N 4 . 56

DRIVE()	N 4 . 57
ECHO	N 4 . 58
END	N 4 . 59
EQUATE	N 4 . 60
EXECUTE	N 4 . 61
EXP	N 4 . 62
EXPENDABLE	N 4 . 63
EXTRACT	N 4 . 64
FIELD	N 4 . 65
FIELDSTORE	N 4 . 67
FLUSH	N 4 . 68
FMT	N 4 . 69
FOOTING	N 4 . 71
FOR ... NEXT	N 4 . 72
FUNCTION	N 4 . 74
GARBAGECOLLECT	N 4 . 76
GETBREAK, GETECHO, GETPRINTER, GETPROMPT	N 4 . 77
GET.CURSOR	N 4 . 78
GOSUB	N 4 . 79
GOTO	N 4 . 80
HASH	N 4 . 81
HEADING	N 4 . 82
ICONV	N 4 . 83
ICONV Boolean (B)	N 4 . 85
ICONV Date (D)	N 4 . 86

ICONV Datetime (DT)	N 4 . 88
ICONV MX, HEX, MO, MB	N 4 . 90
ICONV Masked Decimal (MD)	N 4 . 92
ICONV Masked Scientific (MS)	N 4 . 93
ICONV Time (MT)	N 4 . 94
IF	N 4 . 96
IF (Multiline)	N 4 . 97
IF (Conditional Expression)	N 4 . 98
INDEX	N 4 . 99
INITDIR	N 4 . 100
INITRND	N 4 . 101
INMAT()	N 4 . 102
INP	N 4 . 103
INPUT	N 4 . 104
INSERT	N 4 . 105
INT	N 4 . 107
LEN	N 4 . 109
LINEMARK	N 4 . 110
LOCATE	N 4 . 111
LOCATE BY	N 4 . 113
LOCK	N 4 . 114
LOOP	N 4 . 117
MAT	N 4 . 119
MATCHES	N 4 . 121
MATPARSE	N 4 . 122
MATREAD	N 4 . 123

MATUNPARSE	N 4 . 125
MATWRITE	N 4 . 126
MOD	N 4 . 128
NEG	N 4 . 129
NOT	N 4 . 130
NULL	N 4 . 132
NUM	N 4 . 133
ONCONV	N 4 . 134
ONCONV Boolean (B)	N 4 . 135
ONCONV Date (D)	N 4 . 136
ONCONV Datetime (DT)	N 4 . 138
ONCONV MX, HEX, MO, MB	N 4 . 140
ONCONV Masked Decimal (MD)	N 4 . 142
ONCONV Masked Scientific (MS)	N 4 . 144
ONCONV Time (MT)	N 4 . 145
ON GOSUB	N 4 . 146
ON GOTO	N 4 . 148
OPEN	N 4 . 149
OSBREAD	N 4 . 150
OSBWRITE	N 4 . 152
OSCLOSE	N 4 . 153
OSDELETE	N 4 . 154
OSOPEN	N 4 . 155
OSREAD	N 4 . 156
OSWRITE	N 4 . 158
OUT	N 4 . 159

PAGE	N 4 . 160
PCPERFORM	N 4 . 160
PERFORM	N 4 . 161
PRINT	N 4 . 162
PRINTER	N 4 . 164
PROMPT	N 4 . 165
PUT.CURSOR	N 4 . 166
PWR	N 4 . 167
QUOTE	N 4 . 168
READ	N 4 . 169
READNEXT	N 4 . 171
READNEXT BY	N 4 . 172
READO	N 4 . 175
READV	N 4 . 177
REMOVE	N 4 . 178
REPLACE	N 4 . 179
RETURN/RETURN TO	N 4 . 181
RND	N 4 . 183
SELECT	N 4 . 184
SELECT BY	N 4 . 185
SEQ	N 4 . 188
SERIAL()	N 4 . 189
SIN	N 4 . 190
SPACE	N 4 . 191
SQRT	N 4 . 192
STATUS()	N 4 . 193

STOP	N 4 . 194
STR	N 4 . 195
SUBROUTINE	N 4 . 196
SUM	N 4 . 198
SUSPEND	N 4 . 199
SWAP	N 4 . 201
TAN	N 4 . 202
TIME()	N 4 . 203
TIMEDATE()	N 4 . 204
TRANSFER	N 4 . 204
TRIM, TRIMF, TRIMB	N 4 . 207
UNLOCK	N 4 . 208
WRITE	N 4 . 210
WRITEV	N 4 . 212
XLATE	N 4 . 213



, !, REM, / ... */ (Comments)

Comments are used to explain or document a program. A valid comment statement must begin with either an * (asterisk), an ! (exclamation mark) or REM. Multiline comment blocks can be created using /* and */.

```
[ *
  !
  REM ] {characters}
/* {characters} */
```

Internal Documentation

Comments are a form of internal documentation that allow placing remarks in a program that describe its internal workings without affecting its execution. These comments help to clarify and explain the process or identify other pertinent information.

Good programming includes liberal use of comments throughout a program to document intent or function of the program or a routine within it.

A comment is designated by either the letters REM (for remark), an asterisk (*), or an exclamation mark (!) at the beginning of the statement. The exclamation mark (!) will be printed as a full line of asterisks if you print the program using BLIST. Any characters may be used in a comment statement, including null and blank spaces.

A comment can be written anywhere in a program except before the first line of a program that begins with EXPENDABLE, FUNCTION, or SUBROUTINE.

The text of comments does not affect the size of the object code. However, each line of a comment adds one linemark (one character) to the compiled program.

Single-line Comments

The comment operators REM, * and ! indicate single-line comments. All characters from the comment operator to the end of the physical line are ignored by the compiler.

Comments can be placed on the same physical line as other R/BASIC statements by using a semicolon (;) to separate the two statements. In this case, all characters to the end of the line are ignored, even if there are additional statements separated by a semicolon. For example:

```
FOR CTR = 1 TO 10 ; * loop through 10 times
```

Multi-line Comments

You can create comments that span several lines by using the comment operators /* (to open a comment block) and */ (to close the block). Any number of lines can come between the comment operators. The compiler will ignore all text between the multiline comment operators.

For example:

```
/*  program:  FIELD.REPORT
   author:    John Doe
   date:      01/01/89
*/
```

Correct Use of Comment Statements

```
! data (shelf #, type, title, composer, artist)

IF C = " " THEN GO 3 ELSE GO 5; * test for space between words

/*      All the elements of CUST (I) from 1 to the highest
       key are written to the output file
*/
```

Incorrect Use of Comment Statements

```
X = ! Radius of a circle (r)
A single-line comment appears within another statment.
```

; (Multiple Statement Operator)

Use a semicolon between statements on a single line in an R/BASIC program.

statement ; statement

statement

Any complete R/BASIC statement can precede or follow the semicolon. The comment operators REM, *, and ! at the beginning of a line, however, will mark the entire line as a comment.

Statements separated by semicolons must be complete. Semicolons cannot be placed within a statement.

Careful use of the semicolon operator can increase a program's clarity. It is often used to include a brief comment on a line with executable code.

Correct Use of ; (Multiple Statement Operator)

```
PRINT 'Enter the last name :'; INPUT NAME
```

The INPUT statement appears on the same line as the PRINT statement.

```
PRINT @(-1) ; * CLEAR THE SCREEN
```

The comment is written on the same line as the PRINT statement. The comment must be on the last statement on the line.

[] (Bracket)

The [] (bracket) operator extracts a substring from a string or assigns a substring to a string. The following pages describe how to use the brackets to extract a substring and how to assign a substring.

```
variable[start, length]
variable[start, length] = expression
```

Extracting a Substring

To define a substring, indicate the start position (*start*) of the substring and the length (*length*) of the substring within a complete string. For example, where:

```
S = 'THIS IS A LONG STRING'
```

and where:

```
X = S[6,9]
```

the variable X is defined as the substring of S beginning with the sixth character and continuing for nine characters. In effect, X would equal IS A LONG.

start

Note that each space and character are counted. If the *start* is a positive number, the substring will start at the character in that position counting from the start of the string. If the *start* is a negative number, the substring will start at the character in that position counting from the end of the string. If the *start* is 0 (zero) or 1 (one), the substring will start at the first character in the string.

length

The *length* normally is a positive integer. A *length* equal to 0 (zero) would not extract any characters, and a *length* equal to a negative number would print the substring backwards. For example, if in the example above the *length* were -4:

```
S = S[14,-4]
```

then the extracted substring would begin with the fourteenth character from the beginning and count back four. The result would also print backwards, producing the substring GNOL.

If the second expression starts with an F, then the string is searched forward until a character is found that matches the character following the F. In this example:

```
FORWARD = STRING[3,"FL"]
```

FORWARD is assigned the substring IS IS A because [] (bracket) starts at position 3 and searches forward until an L is found. L is assumed to be a delimiter. Any

valid character can follow the F. COL2() is set to the character position of the delimiter. In this case COL2() has a value of 11. The F is optional if the second expression is a single non-numeric character. The previous example could be written as:

```
FORWARD = S[3, "L"]
```

If the second expression starts with a B, then the string is searched backwards until a character that matches the character following the B is found. In this example:

```
BACK = STRING[14, "BH"]
```

BACK is assigned the substring IS IS A LONG , because [] (bracket) starts at position 14 and searches backwards until an H is found. H is assumed to be a delimiter. Any valid character can follow the B. COL1() is set to the character position of the delimiter H, in this case 2.

Length may also be a system delimiter.

☞ COL1() and COL2() are only set when the *length* is not numeric.

For more information, see COL1() and COL2().

Correct Use of [] (bracket) Operator

```
A = "ABCD0123456789"
```

```
PRINT A[4, 5]
```

Prints "D0123"

```
PRINT A[99, 4]
```

Prints "" (null)

```
PRINT A[-1, 1]
```

Prints "9"

```
PRINT A[-4, 2]
```

Prints "67"

```
PRINT A[6, -3]
```

Prints "10D"

```
PRINT A[-4, -5]
```

Prints "65432"

```
PRINT A[2, "F5"]
```

Prints "BCD01234"

```
PRINT A[10, "BB"]
```

Prints "CD012345"

Assigning a Substring to a String

In addition to extracting a substring of characters from a string, [] (bracket) will also replace a string or substring of characters with another string. To do so, you must identify the characters to replace and the expression with which they will be replaced.

Identify the characters to replace in the variable with the *start* and *length* between brackets.

variable

The *variable* contains the string that is the object of the assignment. That is, you will replace all or part of the *variable* with the expression given in the syntax.

start

The *start* identifies the character position at which the string to replace begins. The *start* must be a number or an expression that yields a number. If the *start* is a positive number, the substring will start at that spot in the character string. If the *start* is a negative number, the substring will start at that number of characters from the end of the string. If the *start* is 0 (zero), the substring will start at the first character in the string. For example:

```
STRING = "ABCDEF"  
STRING[-4, 2] = "XXYY"  
PRINT STRING
```

In this example, ABXXYYEF would be printed. The replacement begins four characters from the end and replaces two characters in the variable with a four character string.

length

The *length* identifies the number of characters from the *start* to replace. The *length* must also be a number or an expression that yields a number. Normally, *length* would be a positive integer. If it were 0 (zero), the expression would be inserted in the variable without replacing characters. If it were negative, the resulting substring would be in reverse order from the original string. For example:

```
STRING = "ABCDEF"  
STRING[4, -3] = "123"  
PRINT STRING
```

would print "A123EF". The replacement begins four characters from the beginning of STRING and ends one character from the beginning.

expression

The *expression* yields a string that replaces the substring in the variable. If the *expression* yields a string that is longer than the specified substring, then the total

string length in the variable is increased. If the *expression* yields a shorter string, then the total string length decreases.

Correct Use of [] (bracket) Operator

```
PRINT @(-1):
LOOP
  PRINT ; PRINT
PRINT 'Enter the variable string':
  INPUT STRING
UNTIL STRING = 'END'
  PRINT 'Enter the start number':
  INPUT START
  PRINT 'Enter the length number':
  INPUT LENGTH
  PRINT 'Enter the expression string':
  INPUT EXPRESSION
  PRINT 'Before ':STRING
  PRINT STRING:' [' :START:',':LENGTH:] = ':EXPRESSION
  STRING[START,LENGTH] = EXPRESSION
  PRINT 'After ':STRING
REPEAT
END
```

The program will request the specifications for a bracket assignment operation and display the results.

Incorrect Use of [] (bracket) Operator

```
WORD[3, "BA"] = "SYL"
```

Uses alpha characters in the specification.

{ } (Braces)

The { } (braces) operators allow you to use a dictionary record to return a value to your program. The braces operators, acting as functions, return the value to their place.

{dictionary.record.key}

dictionary.record.key

The system variables @DICT, @ID, and @RECORD must be set before braces can be used. The braces use the dictionary name, the current record key, and the current record to extract data from a field.

When the system variables are set, enter an existing dictionary record key in the braces.

The braces are useful for program readability. They are also useful when the dictionary key is known and the field number may be changed. When the dictionary key is

not known the `CALCULATE` function is used. When the field number will not change and if speed is important, use `<>` (dynamic array operators).

For more information, see `<>` (Angle Bracket Operators) and `CALCULATE`. For information on using braces within a program, see "Interfacing Dictionaries and R/BASIC and Special R/BASIC Variables" in the chapter "Programming with R/BASIC".

Correct Use of { } (braces)

```
* compare <>, { }, and CALCULATE for field extraction
*
OPEN 'CUSTOMERS' TO CST.FILE ELSE STOP 'No CUSTOMERS file'
OPEN 'DICT','CUSTOMERS' TO @DICT ELSE STOP 'No DICT file'
*
PERFORM 'SELECT 3 CUSTOMERS (S'
*
PRINT @(-1) ; * clear screen
TRUE = 1 ; FALSE = 0
DONE = FALSE
FIELD.NUMBER = 1
FIELD.NAME = 'COMPANY'
*
LOOP READNEXT @ID ELSE DONE = TRUE
UNTIL DONE DO
  READ @RECORD FROM CST.FILE, @ID ELSE STOP 'No record'
  PRINT 'Record      ':@RECORD
  PRINT '{ }        ':{COMPANY}
  PRINT 'CALCULATE  ':CALCULATE(FIELD.NAME)
  PRINT ' <>        ':@RECORD.NUMBER<FIELD.NUMBER>
REPEAT
STOP
```

The program opens the dictionary to @DICT and selects three records from the file. The selection messages are suppressed with the S option. The variables TRUE, FALSE, DONE, FIELD.NUMBER, and FIELD.NAME are set. The LOOP reads each key in the list generated by the SELECT command. When the variables @DICT, @ID, and @RECORD are assigned, the { } can be used. The contents of the field are displayed using the three methods. Note that { } require an existing dictionary record and that CALCULATE and <> can be used with variables. Note also that { }, CALCULATE and <> return the same value to the program.

\$INSERT

The `$INSERT` statement directs the compiler to insert R/BASIC source code from another record into the source code that is currently being compiled.

```
$INSERT {file.name,} record.name
```

Using \$INSERT

The \$INSERT statement is particularly useful for inserting code that is common to more than one program. It is often used with COMMON statements to define COMMON, or to define COMMON in a set of routines. The COMMON statements are placed in a separate record and each subroutine that needs COMMON includes this record.

\$INSERT statements may be nested, but nesting is not recommended because the practice may cause confusion.

Program line numbers referenced by compile and runtime errors are not affected by the inserted source code.

☞ The 32,767 character limit to program size includes inserted source code.

file.name

The *file.name* is the name of the Advanced Revelation file that contains the source code to be inserted. If the *file.name* is not specified, the compiler will search the file from which the program is being compiled. The *file.name* must be the literal filename (not a Qfile or synonym) and may not be surrounded by quotation marks.

record.name

The *record.name* is the literal name. It specifies the name of the record whose source code is to be inserted into the program that is currently being compiled. The *record.name* may not be enclosed in quotes.

Correct Use of \$INSERT Statement

```
$INSERT COM
```

The source code named COM from the current file would be inserted at this point in the program.

```
$INSERT BP,COMMON
```

The record COMMON in file BP would be inserted.

```
$INSERT INITIALIZE
```

The code named INITIALIZE is to be inserted in the program.

Incorrect Use of \$INSERT Statement

```
$INSERT "ARR"
```

A literal string is specified where none is allowed.

```
$INSERT
```

No record name or filename is specified.

+++, --, ***, ///, ::: (Special Multivalue Arithmetic Operators)

The five operators ++, --, ***, ///, and ::: allow addition, subtraction, multiplication, division, and concatenation of two dynamic arrays.

expression operator expression

expression

The *expression* is any R/BASIC expression that yields a dynamic array.

☞ The array operators can only be used on arrays of equal numbers of elements.

The operation is performed using the highest delimiter of the array. If an array has field, value, and subvalue marks, the field marks are used.

The operation compares each of the highest elements of the two arrays from the beginning to the end.

For example, if ++ is used to add two multivalued strings, each value in the first string is added to the value in the corresponding position in the second string. If the number of highest elements is unequal, Advanced Revelation will add 0 (zero) to the unmatched positions. For example:

```
A = 12121212
B = 222222222222
PRINT A++B
```

The result will be 3²3²3²3²22. Zero (0) is added to the fifth multivalue.

Correct Use of Arithmetic Operators

```
* arithmetic array operators
*
PRINT @(-1) ; * Clear the screen
LOOP
  PRINT 'Enter the ASCII number for delimiter A ':
  INPUT A
UNTIL A = 'END'
  PRINT 'Enter the ASCII number for delimiter B ':
  INPUT B
  ADELIM = CHAR(A) ; BDELIM = CHAR(B)
  ARRAY1 = 1:ADELIM:2:ADELIM:3
  ARRAY2 = 10:BDELIM:20:BDELIM:30:BDELIM:40
  ARRAY3 = 100:BDELIM:101:BDELIM:103:ADELIM:200
  ARRAY3 := BDELIM:201:BDELIM:202:ADELIM:300
  PRINT 'ARRAY1 : ':ARRAY1
  PRINT 'ARRAY2 : ':ARRAY2
  PRINT 'ARRAY3 : ':ARRAY3
  * change the operator for more examples
```

```

TOTAL1 = ARRAY1 +++ ARRAY2
TOTAL2 = ARRAY2 +++ ARRAY3
PRINT 'TOTAL1 : ':TOTAL1
PRINT 'TOTAL2 : ':TOTAL2
REPEAT
STOP

```

The dynamic arrays are used in arithmetic operations. The delimiters for field, value, and subvalue correspond to the ASCII numbers 254, 253, and 252.

< > (Angle Bracket Operators)

The angle bracket operators (< >) extract and replace data in dynamic strings.

```

variable
variable<field>
variable<field, value>
variable<field, value, subvalue>

```

< >

Use the < > (angle bracket operators) to replace data in dynamic string arrays, or extract data from them.

The angle bracket syntax is operationally the equivalent of EXTRACT and REPLACE. The angle bracket syntax is more concise and is generally more readable.

Variable

The *variable* designates the dynamic array that contains the data to be extracted or replaced. The *field*, *value*, and *subvalue* expressions specify the location of the data to be extracted or replaced. An expression may be any legal expression that yields a number.

If the angle bracket syntax appears on the left side of an assignment statement, then a dynamic REPLACE will occur. For example:

```
CUST.REC<3,2> = 'JEFFERSON'
```

is equivalent to

```
CUST.REC = REPLACE(CUST.REC, 3, 2, 0, 'JEFFERSON')
```

In this example, the third field, second value is replaced with the string "JEFFERSON". Notice that the 0 (zero) is required in the REPLACE syntax, but not in the angle bracket syntax. Also notice that there is no space between the variable name and the first angle bracket.

- ☛ The variable must be initialized before assigning a first value with angle bracket operators

If the angle bracket syntax is used in any expression, then an EXTRACT is implied. Notice that:

```
NAME = REC<4>
```

is equivalent to

```
NAME = EXTRACT (REC, 4, 0, 0)
```

For more information, see EXTRACT and REPLACE.

Correct Use of Angle Bracket Operators

```
INV.DT = MASTER<3, N>
```

Extract the third field, Nth value.

```
PROD = MASTER<5, N, LINE + 1>
```

Extract the fifth field, Nth value and the subvalue number that is yielded by LINE + 1.

```
MASTER<6, N> = PM<2>
```

Replace the sixth field, Nth value of MASTER with the second field of PM.

```
MASTER<4> = 'WALL CONSTRUCTION'
```

Replace field four of MASTER with "WALL CONSTRUCTION".

Incorrect Use of Angle Bracket Operators

```
IF X<3, 1 THEN STOP
```

>is missing.

```
P<1, 2, 3, 4> = 1
```

Four expressions are used where only three expressions are allowed.

= (Assignment Statement)

An assignment statement is used to assign a value to a variable identifier. It always contains an = (equal sign), which is used as the assignment operator.

```
variable = expression
```

variable

The *variable* always appears on the left side of the equal (=) sign, with the expression that defines its value on the right side of the equal sign.

expression

The value of the *expression* becomes the current value of the variable identifier. The *expression* may be any constant or a calculation from a simple or complex expression.

The *expression* may yield any data type, including numeric, character string, logical, relational or null.

In this example:

```
XYZ = 1000
N = (XYZ+50) *2
```

The value of 1000 is assigned to the variable identifier XYZ in the first statement and N is assigned the value 2100 (1000 + 50) times 2 in the second statement .

In the following example, a string value is assigned:

```
TUNE = "STRING ALONG WITH ME"
```

Correct Use of Assignment Statement

```
Z = 9
```

Assigns 9 to variable identifier Z.

```
A = A - 1
```

Decrement the value of A by one.

```
N = "W.H. STEVENS"
```

Assigns the value "W.H. STEVENS" to N.

```
BOGEY = ""
```

A null value is assigned to BOGEY.

```
MATRIX(A,F) = G(2)
```

Assigns the second element of G to the appropriate element of MATRIX.

Incorrect Use of Assignment Statement

```
NUM 35
```

The assignment operator is missing.

```
"PRICE" = P
```

The variable identifier appears on the right side of the operator.

=, +=, -=, := (Special Assignment Operators)

Four forms of special assignment operators may be used in assignment statements to assign a value to a variable. These are: general assignment, addition assignment, subtraction assignment, and concatenation assignment.

```
variable = expression
variable += expression
variable -= expression
variable := expression
```

variable = expression

When the general assignment operator is used, a value is assigned to a variable without any other operator specified or implied. Assign a variable using a general assignment before using a special assignment. Special assignment operators may not be used with dynamic arrays. No space is allowed between the two operators in a special assignment operation.

variable += expression

This statement is equivalent to:

```
variable = variable + expression
```

The variable is used as a base for further operations to the expression value. In this form, the base variable is added to the expression.

variable -= expression

This statement is equivalent to:

```
variable = variable - expression
```

The expression is subtracted from the base variable.

variable := expression

This statement is equivalent to:

```
variable = variable : expression
```

The expression is concatenated with the base variable.

- ☞ The value of a variable is changed only after the expression has been evaluated.

Correct Use of Assignment Operators

`ANS = 1000/25`

The variable identifier ANS is assigned the value 40.

`T += 3`

T is assigned a value of T plus 3.

`T -= 3`

T is assigned a value of T minus 3.

`T := "AMOUNT DUE"`

AMOUNT DUE will be concatenated onto the end of the value of T.

Incorrect Use of Assignment Operators

`TOTAL /= C`

An attempt has been made to use a division operator.

`TOTAL =+ C`

The order of operators is incorrect.

`REC<3, 2> += 1`

The special operator is used with a dynamic array.

`TOTAL + = C`

The spaces between the + and = operators are not allowed.

=, +=, -=, := (Assigning Matrix Elements)

An assignment statement assigns a value (=, +=, -=, :=) to an individual matrix.

<code>matrix(subscript)</code>	<code>=</code>	<code>expression</code>
<code>matrix(subscript)</code>	<code>+=</code>	<code>expression</code>
<code>matrix(subscript)</code>	<code>-=</code>	<code>expression</code>
<code>matrix(subscript)</code>	<code>:=</code>	<code>expression</code>

Variable Identifier

As in a simple assignment statement, the variable identifier must be placed on the left side of the equal sign. The expression that defined its assigned value must be placed on the right side of the equal sign.

The *matrix* name and the *subscript* that specifies the location of the element within the matrix must appear as a variable on the left side of the equal sign. In the following example:

```
PRICE(75) = R/Q
```

the 75th element in the one-dimensional matrix PRICE is assigned the current value of R divided by the current value of Q. In the following example:

```
CUST(2,3) = "E.T.BARENT"
```

the value of the literal string "E.T.BARENT" is assigned to the element in the second row, third column of the matrix CUST. An element in a matrix may be used as a base for further operations to the expression value, using the special assignment operators. The following formats apply:

```
matrix(subscript) += expression
matrix(subscript) -= expression
matrix(subscript) := expression
```

For more information, see =, +=, -=, := (Special Assignment Operators) and MAT.

Correct Use of Assignment Operators

```
MASTER2(6,8) = CON(3)
```

The value of the third element of matrix CON is assigned to be the value of the element in the sixth row, eighth column position in the matrix named MASTER2.

```
INCOM(8) += MON(30,8)
```

The value of the eighth element of matrix INCOM is increased by the value of the element found in the 30th row, column eight of MON.

```
CUST1(7) := "INC."
```

The literal string "INC". is concatenated to the value found in the seventh element of matrix CUST1.

Incorrect Use of Assignment Operators

```
MATRIX(3,2 = d)
```

The equal sign is included in the parentheses.

```
ARRAY:"TOTAL"
```

The subscripts are missing and an equal sign must be used.

@

The @ function controls the position of the cursor on the monitor, or it performs special print functions.

@ allows you to position the cursor at a particular column and line in the CRT terminal screen. The @ function should not be directed to the printer.

When the @ function is executed, special control codes are activated. These control codes are unique to the specific monitor or microcomputer as defined in the operating system. Compatible computers will function as documented. (Cursor control on other terminals may be executed by PRINT statements with appropriate control codes.)

column

Column places the cursor at a specified column on the current line.

Special Cursor Control Codes:

@(-1)	will clear the screen.
@(-2)	will send the cursor home.
@(-3)	will clear to the end of the screen.
@(-4)	will clear to the end of the line.

line

Line positions the cursor on a specified line of *column*. The *line* and *column* arguments are delimited by a comma.

The values used in *column* and *line* must be within the column and line limits of the monitor, or the cursor position is unpredictable.

:

The colon suppresses the line feed that is normally generated with a PRINT statement, and is necessary for printing at a specified column and line position.

Correct Use of @ Function

```
A = 12
B = 4
PRINT @(A, B) : "Apex Corporation"
Apex Corporation is printed beginning in column 12 on line 4.
```

```
X = 7
PRINT @(X, X+5) : X
The value 7 is printed on line 12 beginning at column 7.
```



```
F = "next screen"
PRINT @(65,23) :F
```

Prints "next screen" beginning at column 65 on line 23.

```
PRINT @(5,5):
INPUT REPLY
```

The cursor will be placed at column 5, row 5, and will wait for input.

```
PRINT @(-1)
```

The screen is cleared.

Incorrect Use of @ Function

```
PRINT @(100,75) : Z
```

Values are out of range.

```
PRINT @(45,-7)
```

Uses a negative expression for row.

ABORT

The ABORT statement forces a return to The Command Level (TCL).

ABORT is used in place of STOP. It guarantees that any and all processing that is in progress will terminate to the process from which it was executed. For more information, see STOP.

ABORT {ALL}

ALL

A system reset is performed when you specify the ALL option.

Correct Use of ABORT

```
ABORT
```

Return to TCL.

```
IF X EQ 'END' THEN ABORT
```

Control returns to TCL if variable X is equal to 'END'

Incorrect Use of ABORT

```
ABORT X
```

Uses an expression where none is allowed.

ABORT 'PROGRAM ABORTED'
Attempts to print after the ABORT.

ABS

The ABS function returns the unsigned numeric value of an expression.

ABS generates the absolute numeric value (that is the unsigned value) of an expression. The result looks like a positive value but technically it is unsigned.

ABS (expression)

expression

The *expression* always returns a numeric value. For example, in the following expression, if the variable X has the value of 40, then

```
X = 40
Y = ABS (10-X)
```

assigns the value of 30 to variable Y.

- ☞ The absolute value of a real number is its distance from 0 (zero) on the number line. The absolute value of a number is never negative. It is the distance from 0 (zero) without regard to which direction it is from 0 (zero).

Correct Use of ABS

```
PRINT ABS (-100)
```

Prints the absolute value of the constant -100.

```
A = 500
B = ABS (A-800)
```

Assigns the value 300 to the variable B.

Incorrect Use of ABS

```
X = "BALANCE"
Y = ABS (X)
```

Uses a non-numeric expression.

```
ABS (J) = 100
```

The function appears on the left side of the assignment operator.

ALPHA

The ALPHA function logically determines the presence of a string of alphabetic characters and returns a 1 (true) if the string is not numeric or a 0 (false) if the string is numeric.

ALPHA is the opposite of NUM.

ALPHA (expression)

expression

If the *expression* contains any non-numeric characters including punctuation, then a 1 (true) is returned. If the expression contains a numeric value, then a 0 (false) is returned.

For more information, see NUM. Also see “Numeric Constants” in the chapter “Programming with R/BASIC”.

Correct Use of ALPHA

```
PROMPT "
LOOP
  PRINT @(10, 4):
  INPUT INPUT.STRING
UNTIL INPUT.STRING = 'END'
  IF ALPHA(INPUT.STRING) THEN
    PRINT @(10, 0):'Alpha Character(s)':@(-4)
  END ELSE
    PRINT @(10, 0):'All Numeric':@(-4)
  END
PRINT
REPEAT
END
```

The program will test the input string for an alpha character. “All Numeric” is printed if there are no alpha characters.

Incorrect Use of ALPHA

```
ALPHA(X) = 1
```

The function appears on the left side of the assignment operator.

AND and OR

A logical expression results when two or more propositions (expressions) are separated by a Boolean logic operator AND or OR. Logical expressions evaluate to 1 (one) if true, 0 (zero) if false.

proposition

AND

OR

proposition

Logical Expression

A logical expression consists of two or more propositions that are separated by either of the logical operators AND or OR. Logical operators have the lowest priority of all operators and are evaluated only after all math, string, and comparison operations given in each proposition have been executed.

AND

In a simplified truth table following, p represents one proposition, and q the second proposition. For instance, if both the p AND q propositions of a certain logical expression are evaluated to be true, the computer will evaluate the logical expression to 1 (true).

OR

In another case, if p is evaluated false OR q is evaluated true, the logical expression will evaluate to 1 (true) also.

For more information, see NOT.

Simplified Truth Table

AND		
p	q	p AND q
T	T	T
T	F	F
F	T	F
F	F	F

OR		
p	q	p OR q
T	T	T
T	F	T
F	T	T
F	F	F

Correct Use of Logical Expressions

```
IF (TIME > 63) AND (POINTS GE 24) THEN
  PASS = 0
END
Do not assign pass a value of 0 unless the conditions are met.

A = 4
B = 5
C = 0
D = A > 3 AND B LE 5
E = A AND B
F = (A OR C) AND (B AND C)
D is set to 1 (true). E is set to 1 (true). F is set to 0 (false).
```

ATAN

The ATAN function generates the trigonometric arctangent of an angle.

ATAN (expression)

expression

The *expression* must be a numeric value that designates the tangent of an angle.

Correct Use of ATAN

```
PRINT 'TANGENT', 'ATAN(TANGENT)'
FOR TANGENT = -5 TO 5 STEP 0.5
  PRINT TANGENT, ATAN(TANGENT)
NEXT TANGENT
END
produces these results:
```

TANGENT	ATAN
-5	-78.6901
-4.5	-77.4712
-4	-75.9638
-3.5	-74.0546
-3	-71.5651
-2.5	-68.1986
-2	-63.4349
-1.5	-56.3099
-1	-45
-.5	-26.5651
0	0
.5	26.5651
1	45
1.5	56.3099

TANGENT	ATAN (Continued)
2	63.4349
2.5	68.1986
3	71.5651
3.5	74.0546
4	75.9638
4.5	77.4712
5	78.6901

ARCTAN = ATAN(50)
The variable identifier ARCTAN will be set to the arctangent of 50.

Incorrect Use of ATAN

Y = ATAN()
The expression is missing.

T = ATAN[X]
Uses brackets where parentheses are required

BITAND, BITOR, BITXOR, BITNOT

The BITAND, BITOR, BITXOR, and BITNOT functions perform bit-wise AND, OR, XOR, and NOT logical functions on two numbers.

BITAND
BITOR (expression, expression)
BITXOR
BITNOT

These functions can be used to calculate the binary result from subjecting two integer numbers to AND, OR, XOR, and NOT evaluation. The result will be an integer. Each number may be up to 64 bits in length. Hence 2 to the 64th (approximately 10 to the 20th) is the largest number that can be used by these functions.

These functions can be useful when using the OUT statement and the INP function.

BITAND

BITAND takes two integers and calculates the bit-wise logical AND of the two numbers.

BITOR

BITOR takes two integers and calculates the bit-wise logical OR of the two numbers.

BITXOR

BITXOR takes two integers and calculates the bit-wise logical exclusive OR of the two numbers.

BITNOT

BITNOT takes the decimal integer yielded by expression and calculates the bit-wise logical NOT of that number.

expression

The *expression* must yield an integer.

Correct Use of BIT Functions

EXPRESSION1	EXPRESSION2	BITAND	BITOR	BITXOR
0	0	0	0	0
1	0	0	1	1
1	1	1	1	0
5	1	1	5	4
63	31	31	63	32

```
LOOP
  PRINT "X1  ":INPUT X1
UNTIL X1 = 'END' DO
  PRINT "X2  ":
  INPUT X2
  PRINT "BITAND:  " : BITAND(X1, X2)
  PRINT "BITOR:   " : BITOR(X1, X2)
  PRINT "BITXOR:  " : BITXOR(X1, X2)
REPEAT
```

The program will print the results of the BIT function tests.

Incorrect Use of BIT Functions

BITAND = 1
A function cannot be assigned a value.

X = BITOR(3)
One expression is missing.

BREAK KEY

The BREAK KEY statement designates whether the break key on the terminal is to be operational or inhibited during the execution of a program.

BREAK {KEY}	ON OFF <i>expression</i>
--------------------	--

ON

Permits the Break key on the terminal to be operational. That is, a terminal user may press [Ctrl-Break] to interrupt the normal flow of the program. When Break is pressed, the Debugger is called.

OFF

Disables the Break key. BREAK OFF prohibits the user from interrupting the normal flow of the program. This can be an important control during a routine, such as a sorting routine, in which it is critical that execution of the program not be disabled.

expression

The *expression* specifies the status of the Break key. If the value of the *expression* is true (non-zero), the Break key is operational. If the value of the expression is false (zero) or null, the Break key is inhibited.

☛ For details on the Debugger, see DEBUG. Also see the GET functions.

Correct Use of BREAK

BREAK OFF

The register is set to 0 (zero) and break will be inhibited.

BREAK KEY TEST1

If the value of TEST1 is true, the Break key is made operational.

CALCULATE

The CALCULATE function is used to interface dictionary words to R/BASIC. CALCULATE returns the result of evaluating the dictionary record.

CALCULATE (<i>expression</i>)

Using CALCULATE

You can use calculate in place of the {} (braces) function. It must be used if the dictionary word is not known at compilation time.

The following three lines are equivalent:

```
PRINT {INV.AMT}
PRINT CALCULATE('INV.AMT')
X = "INV.AMT" ; PRINT CALCULATE(X)
```

Notice that in each case, the dictionary word INV.AMT is executed.

expression

The *expression* must yield a dictionary name in the file that has been opened to the system variable @DICT.

For more information, see "Interfacing Dictionaries and R/BASIC" in the chapter "Programming with R/BASIC".

Correct Use of CALCULATE

```
PROMPT ""
PRINT "Name of file to display:": ; INPUT FILE.NAME
OPEN FILE.NAME TO FILE ELSE STOP "No such file"
OPEN "DICT.":FILE.NAME TO @DICT ELSE STOP "No DICT"
*
PRINT "Field #1 to display:": ; INPUT F1
PRINT "Field #2 to display:": ; INPUT F2
PRINT "Field #3 to display:": ; INPUT F3
*
HEADING "Sample field listing for ":FILE.NAME:"'L'"
SELECT FILE
EOF=0
LOOP
  READNEXT @ID ELSE EOF=1
UNTIL EOF
  READ @RECORD FROM FILE, @ID ELSE
    STOP "No record @ ":@ID
  END
  PRINT @ID, CALCULATE(F1), CALCULATE(F2), CALCULATE(F3)
REPEAT
STOP
```

Incorrect Use of CALCULATE

```
CALCULATE(X) = 3
```

The function appears on the left side of the assignment operator.

```
Z = CALCULATE(X, Y)
```

Uses more than one expression.

CALL

The CALL statement allows branching to an external subroutine. Program control can be transferred to a subroutine that has been compiled and cataloged separately from the program or programs that call it.

CALL	$\left[\begin{array}{l} \text{subroutine} \\ @variable \\ @matrix(index) \end{array} \right]$	{argument [{, argument} ::]}
------	--	-------------------------------

subroutine

The CALL *subroutine* format transfers program control directly to the specified subroutine. The *subroutine* must have a catalog pointer in the VOC file that is currently attached. The name must be written exactly as it was specified when the *subroutine* was cataloged.

Control will return to the calling program on the line following the CALL statement. For information about cataloging programs, see the MAKEVOC command in the “TCL Command Reference” chapter of *TCL*.

argument

An argument may be a variable, matrix, constant or expression that is used to pass values to and from the calling program and the subroutine. The calling program must use the same number of arguments as specified in the subroutine to be called.

The *argument* may consist of one or more expressions, separated by commas and enclosed in parentheses. If no arguments are specified, the parentheses are omitted.

Use the @*variable* or @*matrix* formats to indirectly call the external subroutine. The subroutine is called indirectly by first assigning the subroutine as the value of a variable or matrix, and then using the CALL @ statement. In the following example, the subroutine was cataloged as FEDTAX:

```
ITAX = "FEDTAX"
CALL @ITAX
```

The program would call the subroutine FEDTAX. The number of arguments passed must be less than or equal to the number of arguments specified in the SUBROUTINE header in the subroutine.

A main program can be called from another main program. This allows memory to be released when the called program is terminated. In this case, no arguments may be specified.

✎ For more information, see DECLARE, EXECUTE, EXPENDABLE, FUNCTION, PERFORM, RETURN and SUBROUTINE.

Correct Use of CALL

```
CALL PUSH
```

Calls the subroutine named PUSH.

```
SNAME = 'FACTOR'
```

```
CALL @SNAME(Z)
```

Calls the subroutine FACTOR and passes Z as an argument.

```
DIM X(30)
```

```
DIM Q(30)
```

```
FOR Y = 1 TO 30
```

```
  X(Y) = Y
```

```
  Q(Y) = Y
```

```
  PRINT Y
```

```
NEXT Y
```

```
CALL SS(MAT X)
```

```
PRINT "Back to main"
```

```
END
```

```
SUBROUTINE SS(MAT X)
```

```
DIM X(30)
```

```
FOR Y = 1 TO 30
```

```
  PRINT X(Y)
```

```
NEXT Y
```

```
RETURN
```

The calling program passes the matrix X to the cataloged subroutine SS that prints the values in X.

Incorrect Use of CALL

```
CALL VENDOR(X,Z,A,B)
```

```
SUBROUTINE VENDOR(X,Y,Z)
```

Uses mismatched arguments

CASE

The CASE statement provides a framework for conditional branching. It can evaluate one or more test expressions and pass control to selected portions of the program.

A CASE statement must be introduced with BEGIN CASE and terminated with END CASE.

```
BEGIN CASE
```

```
  [CASE expression statements ::]
```

```
END CASE
```

expression

The *expression* is evaluated. If the value is true (non-zero), the *statements* are executed, and no subsequent CASE statements are evaluated. Program control then proceeds past END CASE.

If all the CASE *expressions* are false, program control proceeds past END CASE without executing any of the *statements*.

When used with CASE, an *expression* of 1 (one) always equates to true. It can be used to specify unconditional execution of the statements. If used in this way, this should be the last CASE statement. This CASE will always be executed if none of the other CASE statements evaluate to true.

statements

The CASE *statements* may be any valid R/BASIC statement or statements.

Correct Use of CASE

```
BEGIN CASE
  CASE A = B; GOSUB 100
  CASE A < B; GOSUB 200
  CASE 1; GOSUB 300
END CASE
```

The program control will branch to statement label 100 if A is equal to B, to 200 if A is less than B, and to 300 if neither of the above is true.

```
BEGIN CASE
  CASE QTY < 10
    CALL EXPEDITE
  CASE QTY > 100
    CALL REORDER
END CASE
```

If the inventory quantity shows less than 10, the program calls the subroutine EXPEDITE. If the QTY is more than 10 but less than 100, the program calls the subroutine REORDER.

Incorrect Use of CASE

```
BEGIN CASE
  X = 1
```

No CASE header is used.

```
CASE NULL
```

Uses an invalid expression.

```
END
```

END is used in place of END CASE.

CHAIN

The CHAIN statement passes control to another program, and removes the current program from memory. The program called with CHAIN can be any TCL command or cataloged R/BASIC program.

CHAIN *expression*

expression

The name of the program or command called with CHAIN is passed as the *expression*. The command called with CHAIN must have an entry in the VOC file that is currently attached.

The current program is removed from the program stack, releasing the memory it occupies.

There is no direct method to pass data to a program called via CHAIN. The current data buffer (@DATA) and any pending select list are both cleared. However, global variables, including @USER0-4 and @RECUR0-4, and labelled common areas retain their value and can be used to pass data to the chained program.

Correct Use of CHAIN Statement

```
CHAIN "LISTFILES"
```

Executes the TCL command LISTFILES

```
PROMPT ""
COMMAND = "LIST TEST "
LOOP
  PRINT "Enter field to display ([Enter] to quit):"
  INPUT FIELD_NAME
  UNTIL FIELD_NAME = ""
    COMMAND := FIELD_NAME : " "
  REPEAT
  PRINT COMMAND
  CHAIN COMMAND
  STOP
```

The program builds a TCL LIST command dynamically. After prompting the user for the fields to be displayed, the program chains to the LIST command.

Incorrect Use of CHAIN Statement

```
CHAIN VENDOR (X, Z, A, B)
```

No arguments can be passed to the process called via CHAIN.

CHAR

The CHAR function converts the numeric value of an expression to its corresponding ASCII character string value.

CHAR (expression)

Using CHAR

In the following example, variable PERCENT is assigned the ASCII string value of %. The percent sign (%) is the ASCII equivalent of decimal 37.

```
PERCENT = CHAR(37)
```

In the following example, the ASCII string value of variable identifier C is assigned to AB.

```
AB = CHAR(C)
```

CHAR will be compiled as a special load instruction if the CHAR *expression* is a decimal integer constant. This instruction generates less object code and is executed much faster. In this example:

```
FM = CHAR(254)
```

CHAR assigns a field mark to the variable identifier FM.

expression

CHAR requires that the value of the *expression* must be an integer within the range of 0 to 255.

Correct Use of CHAR

```
X = 254
```

```
FM = CHAR(X)
```

Assigns the string value for a field mark (ASCII character 254) to the variable identifier FM.

```
BEEP = CHAR(7)
```

The BELL character (ASCII character 7) is assigned to BEEP.

```
ESC = CHAR(27)
```

ESC is assigned an escape character (ASCII character 27).

```
PRINT CHAR(35)
```

A # is printed.

```
CRLF = CHAR(13) : CHAR(10)
```

Assigns a carriage return (ASCII character 13) followed by a line feed (ASCII character 10) to CRLF.

Incorrect Use of CHAR

```
X = CHAR("B")
```

Uses a non-numeric expression.

```
A = CHAR(27) , CHAR(2)
```

A comma instead of a colon is used to separate functions.

CLEAR

The CLEAR statement assigns a value of null ("") to all variables that are not in the COMMON area.

CLEAR

Using CLEAR

CLEAR may be used anywhere in the program. When used at the beginning of a program, CLEAR may be used to initialize variables and prevent the possible runtime error of a non-assigned variable.

When used within the body of a program, CLEAR will reassign all previously assigned values, including matrix elements, to null. Values and elements, except those in the COMMON area, are lost when reassigned to null. This includes any parameters passed in the subroutine call.

Correct Use of CLEAR

```
CLEAR
```

Assigns the value of null to all variables that are not in the COMMON area.

Incorrect Use of CLEAR

```
CLEAR X
```

Uses a variable. The CLEAR command does not use a variable.

CLEAR COMMON

The CLEAR COMMON statement assigns a value of null to all variables previously named in the COMMON area. The values of variables that are not in the COMMON area are not affected by this statement.

CLEAR COMMON

Using CLEAR COMMON

Use CLEAR COMMON in a subroutine to reset the common variables to null. CLEAR COMMON may be used in a main program to initialize variables and prevent the possible runtime error of a non-assigned variable.

Labeled COMMON areas are not cleared by the CLEAR COMMON statement.

Correct Use of CLEAR COMMON

CLEAR COMMON

Assigns the value of null to all the variables in the COMMON area.

Incorrect Use of CLEAR COMMON

CLEAR COMMON J

Uses a variable identifier where none is allowed. (The CLEAR COMMON statement assigns all variables in the COMMON area to 0 (zero).)

CLEARDATA

The CLEARDATA statement clears all previous data stored by the DATA statement.

CLEARDATA

Using CLEARDATA

Data that has been previously loaded using the DATA statement may be set to null by using the CLEARDATA statement. CLEARDATA is the equivalent of setting the special variable @DATA to null ("").

Correct Use of CLEARDATA

CLEARDATA

Erases data that was loaded into a record by a DATA statement by setting its value to null.


```

CLEAR
ECHO OFF
DATA 1,2,3,4,5,6,7,8
LOOP
  INPUT X
  Y += X
UNTIL Y > 15 REPEAT
CLEARDATA
PRINT "TOTAL = ":Y

```

The INPUT statement uses the list of numbers supplied by the DATA 15, the list is cleared.

Incorrect Use of CLEARDATA

```
CLEAR DATA
```

Uses a space between CLEAR and DATA.

CLEARFILE

The CLEARFILE statement deletes all records from a file, but retains the file in the system.

CLEARFILE filevar THEN|ELSE statements

filevar

Filevar is a file variable created by a previous OPEN statement.

THEN

The *statement(s)* following THEN are executed if a file is cleared successfully.

☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the file cannot be cleared. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains details about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

☞ For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in a CLEARFILE statement are optional. However, it is strongly recommended that all CLEARFILE logic be constructed to accommodate possible unsuccessful file clears. If no THEN or ELSE clause is specified, any errors encountered during the CLEARFILE operation are passed to the system error handling routine.

Correct Use of CLEARFILE Statement

```
CLEARFILE FILE.CUSTOMER ELSE
  CALL MSG("Unable to clear file!", "", "", "")
END
```

Clears the file whose file variable is in FILE.CUSTOMER. If the file cannot be cleared, a message is displayed to the user.

```
DECLARE SUBROUTINE MSG, FSMSG
FILE = ""
MSG("Enter name of file to clear:", "R", FILE, "")
IF FILE NE "" THEN
  OPEN FILE TO FILE_VAR THEN
    RESP = "N"
    MSG("Are you sure you wish to clear file %1%?", "RC", RESP, FILE)
    IF RESP[1,1] EQ "Y" THEN
      CLEARFILE FILE_VAR THEN
        MSG("File ":FILE:" successfully cleared.", "", "", "")
      END ELSE
        FSMSG("File ":FILE:" cannot be cleared.")
      END
    END ; * if resp eq 'Y'
  END ELSE
    FSMSG()
  END ; * open
END ; * if file then
STOP
```

The program prompts the user for the name of a file to clear. The file is opened, and then cleared.

Incorrect Use of CLEARFILE Statement

```
CLEARFILE "CUSTOMER"
CLEARFILE requires a file variable.
```

CLEARSELECT

The CLEARSELECT statement sets the list of record keys for a specified cursor to null. CLEARSELECT in R/BASIC has the same function as CLEARLIST at TCL.

CLEARSELECT {cursorvar}

cursorvar

If specified, *cursorvar* identifies a cursor previously established with a SELECT ... BY statement. If a *cursorvar* is not specified, the current active select list is cleared (cursor zero). *Cursorvar* can also be an integer value of 0 to 8, indicating the cursor number to clear.

- ☞ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

Correct Use of CLEARSELECT

```
CLEARSELECT
```

Sets the currently active list of record keys to null.

```
FOR CTR = 0 TO 8
  CLEARSELECT CTR
NEXT CTR
```

Clears all cursors.

Incorrect Use of CLEARSELECT

```
CLEARSELECT 9
```

Cursors are numbered 0 to 8.

COL1()

The COL1() function can be used only after the FIELD and [] (bracket) functions. COL1() finds the position of the delimiter preceding the selected substring.

COL1 ()

Using COL1()

After FIELD or [] has selected a specific substring, use COL1() to determine position of the delimiter preceding the substring. In this example:

```
AMT = FIELD("999/888/777", "/", 2)
BEFORE = COL1( )
```

the substring selected by FIELD would be 888. COL1() assigns the numeric value of 4 to BEFORE.

COL1() returns 0 (zero) if the specified substring is not found. A 0 (zero) is also returned if the delimiter expression of the FIELD function is null. If a FIELD or [] function has not been previously executed, the value returned is undefined.

COL1() retains its value until another FIELD or [] function is executed.

- ☞ For more information, see COL2(), FIELD, and [] (bracket).

Correct Use of COL1()

```
P = FIELD("ABRACADABRA", "A", 4)
S = COL1( )
```

The variable identifier P is assigned the string value "D" and the numeric value of 6 is assigned to S.

Incorrect Use of COL1()

```
P = COL1(r)
```

Parameter appears in the parentheses where none is allowed.

```
P2 = COL( )
```

The 1 or 2 after COL is missing.

COL2()

The COL2() function can be used only after the FIELD and [] (bracket) functions. COL2() finds the position of the delimiter following the selected substring.

COL2 ()

Using COL2()

After FIELD or [] has selected a specific substring, use COL2() to determine the position of the delimiter following the substring. In this example:

```
AMT = FIELD("999/888/777", "/", 2)
AFTER = COL2( )
```

the substring selected by FIELD would be 888. COL2() assigns the numeric value of 8 to AFTER.

COL2() returns 0 (zero) if the specified substring is not found. A 0 (zero) is also returned if the delimiter expression of the FIELD function is null. If a FIELD or [] function has not been previously executed, the value returned is undefined.

COL2() retains its value until another FIELD or [] function is executed.

For more information, see COL1(), FIELD, and [] (bracket).

Correct Use of COL2()

```
P = FIELD("ABRACADABRA", "A", 4)
E = COL2( )
```

The variable identifier P is assigned the string value "D", and the value of 8 is assigned to E.

```
CO = FIELD("88D77D33D44", "D", 2)
LOCO = COL2()
```

LOCO is assigned the numeric value of 6.

```
PRINT "Enter date in MM-DD-YY format":
INPUT DATE
TMP = FIELD( DATE, "-", 2)
PRINT "MONTH = ":DATE[1, COL1()-1]
PRINT "DAY = ":DATE[COL1() + 1, COL2() - COL1() - 1]
PRINT "YEAR = ":DATE[COL2() + 1, 2]
END
```

The month, day, and year will be printed without the dashes.

Incorrect Use of COL2()

```
P = COL2(r)
```

Parameter appears in the parentheses where none is allowed.

```
P2 = COL()
```

The 1 or 2 after COL is missing.

COLHEADING

The COLHEADING statement specifies the headings that will appear above each column of data on a report. COLHEADING must be used in conjunction with COLLENGTH. If the COLLENGTH is specified to be shorter than the COLHEADING, the COLHEADING will be truncated at the last character specified in COLLENGTH.

COLHEADING works with the HEADING, FOOTING, and PAGE statements.

```
COLHEADING heading [{ : @FM : heading } ::]
```

heading

The *heading* may be a character string, or an expression yielding the appropriate string format. If more than one *heading* is specified, the headings are delimited by @FM (field marks). Each heading will appear at the top of a column.

Each *heading* in a COLHEADING statement may contain up to 50 characters. Column headings will appear on the first line below any print statement using the HEADING statement. Use @VM (value marks) to produce column headings of more than one line.

- ☞ For more information, see COLLENGTH. See also the TCL LIST command S option in *TCL* for a method to generate programs that use COLHEADING and COLLENGTH.

Correct Use of COLHEADING

```
COLH = "NAME" :@FM: "ADDRESS"
COLHEADING COLH
```

The expression COLH is used to define the headings.

```
* HEADING/FOOTING
HEADING "SHIPPING REPORT"
FOOTING ""
* MAKE COLUMN HEADING
COL.VAR = "SHIP.REQ" : @FM : "DATE" : @FM
COL.VAR := "#" : @FM : "ATT" : @FM : "DESC"
COLHEADING COL.VAR
LEN.VAR = 10 : @FM : 10 : @FM : 5
LEN.VAR := @FM : 30 : @FM : 20
COLLENGTH LEN.VAR
PRINT S.ATID "L#10" : " " :
PRINT S.DATE "L#10" : " " :
PRINT S.SERIAL "L#5" : " " :
PRINT S.ATT. "L#30" : " " :
PRINT S.DESC "L#20"
END
```

The program segment prints a heading and a column heading and shows how fields would be printed. Note that the program is not complete. The variables to be printed are not assigned.

Incorrect Use of COLHEADING

```
COLHEADING "X"
```

The COLLENGTH statement is missing.

COLLENGTH

The COLLENGTH statement specifies the length each column will occupy for a report. The columns must have been initialized with a COLHEADING statement.

```
COLLENGTH length [{ : @FM : length } ::]
```

Using COLLENGTH

Use COLLENGTH with COLHEADING. Each *length* specified in COLLENGTH indicates the length of the associated heading specified in COLHEADING.

If the heading string specified in COLHEADING is longer than the *length* specified in COLLENGTH, the heading will be truncated at the appropriate character. If shorter, the heading will be padded with periods. COLLENGTH does not affect the display of data. To format the display of data, use FMT.

length

The *length* may be a number or any valid expression that yields the length of the column. If more than one column length is specified, each *length* is delimited by an @FM (field mark).

Correct Use of COLLENGTH

```
COLL = 15 : @FM: 25
COLLENGTH COLL
```

COLLENGTH uses the expression COLL.

```
COLHEADING "NAME" : @FM : "PHONE"
COLLENGTH 15 :@FM: 8
```

This would create a report with the following column heading lengths:

Column	Length
NAME	15
PHONE	8

Incorrect Use of COLLENGTH

```
COLLENGTH 15 8 25
```

The @FM delimiters are missing.

COMMON

The COMMON statement provides a common area for storing variables used by both the main program and external subroutines.

```
COMMON variable [{, variable} ::]
(or)
COM  variable [{, variable} ::]
```

Using COMMON

All variables not passed as a parameter that are to be used in both a main program and an external subroutine must be defined by a COMMON statement prior to their use. Matrices must be named and dimensioned by either a COMMON statement or a DIMENSION statement, but not both before a program uses them. The size and number of the dimensions in a matrix will be permanently set the first time the program encounters a COMMON statement that defines it. Subsequent COMMON or DIMENSION statements will not change the dimensions of the matrix.

COMMON allocates variables in the order in which they occur. Variables are passed between programs by reference. The COMMON statement in a called program may use different names than the main program. The first COMMON variable in the main program is referenced by the first COMMON variable in the subroutine, the second variable is referenced by the second variable in the subroutine and so on.

All COMMON area variables initially are unassigned until they are assigned a value during the execution of the program. A CLEAR COMMON statement reassigns all common area variables to a value of null. Common areas are recursive. That is, any main program that is executed by way of TCL or the CALL statement creates a new common area for that main program and its subroutines. Hence, main programs cannot share the same common area. If a COMMON block is used, it must be defined in the main program, even if the main program does not use any of the common variables. Only a main program can create a common block. Subroutines can access COMMON but cannot create it.

✎ For more information, see CLEAR COMMON and Labeled COMMON.

variable

Any valid identifier may be used as a *variable*. At least one *variable* must be specified. The COMMON area may contain up to 255 variables.

Correct Use of COMMON

```
COMMON X, Y, Z (5)
```

Assigns the variable identifiers X and Y to the common area along with the array Z.

```
COM A, B, C
```

```
COM D, E, F
```

Assigns A, B, C, D, E, and F to the common area.

Incorrect Use of COMMON

```
COMMON W, R, T (10)
```

```
DIM T (10)
```

The array T cannot be declared by a DIMENSION statement after it has been declared in a COMMON statement.

```
COMMON F, G, H,
```

```
I, P
```

Continuing lines are not allowed.

Labeled COMMON

The labeled COMMON statement provides a common area for storing variables used by any program that contains a COMMON with the same label.

```
COMMON /label/ variable [ {,variable} ::]
(or)
COM      /label/ variable [ {,variable} ::]
```

Common Variables

Unless passed as a parameter, all variables shared by programs must be defined by a COMMON statement prior to their use. The values in variables defined by Labeled COMMON retain their values until specifically reset or until logoff. The values in variables defined by COMMON are lost when the main program ends. Labeled COMMON is not recursive. When another main program is executed by way of TCL or the CALL statement, the values in the variables are retained. Any legal type of variable identifier may be designated. Matrices must be named and dimensioned by either a COMMON statement or a DIMENSION statement, but not both, before a program uses them. The size and number of the dimensions in a matrix will be permanently set the first time the program encounters a COMMON statement that defines it. Subsequent COMMON or DIMENSION statements will not change the dimensions of the matrix.

/label/

A COMMON statement that contains a *label* is a labeled COMMON statement. The *label* may be any string that does not contain the character /.

COMMON allocates variables in the order in which they occur. Variables are passed between programs by reference to the allocation. The COMMON statement in a called program may use different names than the main program. The first COMMON variable in the main program is referenced by the first COMMON variable in the subroutine, the second variable is referenced by the second variable in the subroutine.

☞ The COMMON area may contain up to 255 variable identifiers.

Correct Use of Labeled COMMON

```
COMMON A,B,C(5)
COMMON /LABEL.COM/ X,Y
PRINT @(-1) ; *CLEAR SCREEN
CRLF = CHAR(13) : CHAR(10)
A=1; B=2; X=3; Y=4
FOR CNT = 1 TO 5
    C(CNT) = CNT * 10
NEXT CNT
CALL COMSUB
PRINT "A = ":A:CRLF:"B = ":B:CRLF:"X = ":X:CRLF:"Y = ":Y
```

```

FOR CNT = 1 TO 5
  PRINT "C(":CNT:") = ":C(CNT)
NEXT CNT
STOP

```

```

SUBROUTINE COMSUB
COMMON COMA,COMB,COMC(5)
COMMON /LABEL.COM/ LCOMX,LCOMY
COMA += LCOMX
COMB += LCOMY
FOR CNT = 1 TO 5
  COMC(CNT) -= COMB
NEXT CNT
RETURN

```

The main program will print the values in the COMMON areas after they have been modified by the subroutine COMSUB.

CONVERT

The CONVERT statement converts characters in a string to other specified characters. It is commonly used to convert strings from uppercase to lowercase or the reverse.

CONVERT substring1 TO substring2 IN string

substring1

Substring1 identifies a substring of *string*. Each character(s) in *substring1* will be replaced by the character(s) in the corresponding position of *substring2* (the *i*th character of *substring1* is replaced by the *i*th character of *substring2*).

substring2

Substring2 is the substring of *string* that contains the replacement character(s) for *string1*.

In the following example, each Q in the variable STRING will be changed to Z; each B will be changed to O; and each U will be changed to X:

```
CONVERT "QBU" TO "ZOX" IN STRING
```

If the number of characters specified in *substring1* exceeds the number of characters specified in *substring2*, the character that has no corresponding replacement specified in *substring2* will be deleted from the *variable*.

In this example:

```

STRING = "GOOD DAY DONNY"
CONVERT "DAY" TO "BY" IN STRING

```

the variable **STRING** will have the value **GOOB BY BONN**. Notice that both **Ds** in **GOOD DAY** are converted to **B's**, the **A** is converted to **Y** and the **Y** is deleted.

variable

A variable identifying the *string* to be converted.

Correct Use of CONVERT

```
CONVERT "N" TO CHAR(35) IN T
```

"N"s will be converted to # (ASCII character 35) in the variable identifier T.

```
LOOP
```

```
  PRINT @(-1)
```

```
  PROMPT ""
```

```
  PRINT "Alpha string to convert:"
```

```
  INPUT DATA.IN
```

```
  L.CASE = "abcdefghijklmnopqrstuvwxyz"
```

```
  U.CASE = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
UNTIL DATA.IN = "END"
```

```
  CONVERT U.CASE TO L.CASE IN DATA.IN
```

```
  FIRST.LET = DATA.IN[1,1]
```

```
  CONVERT L.CASE TO U.CASE IN FIRST.LET
```

```
  DATA.IN[1,1] = FIRST.LET
```

```
  PRINT @(0,2):"Converted string: ": DATA.IN
```

```
  PRINT @(0,21):"Press [Enter] to continue":
```

```
  INPUT XXX ; * pause
```

```
REPEAT
```

```
END
```

The program converts the input string to lowercase, then converts the first letter to uppercase.

Incorrect Use of CONVERT

```
CONVERT "J" TO "KL" IN D
```

The number of characters in substring2 is greater than the number in substring1.

```
CONVERT X TO Z
```

No variable is specified.

COS

The **COS** function calculates the trigonometric cosine of an angle.

COS (expression)

expression

The *expression* must be a numeric value that designates degrees. The values must be within the range 0 to 360.

The following is an example of a program using COS and its resulting output:

```
PRINT "ANGLE", "COS (ANGLE) "
FOR ANGLE = 0 TO 360 STEP 45
    PRINT ANGLE, COS (ANGLE)
NEXT ANGLE
END
```

ANGLE	COS(ANGLE)
0	1
45	.7071
90	0
135	-.7071
180	-1
225	-.7071
270	0
315	.7071
360	1

Correct Use of COS Function

$TAN = SIN(X) / COS(X)$

The sine of X is divided by the cosine of X and assigned to the variable identifier TAN.

Incorrect Use of COS Function

$COS(X) = 5$

The function appears on the left side of the assignment operator.

COUNT

The COUNT function counts the number of times a string is repeated in a source string and returns that number.

COUNT(source_string, search_string)

source_string

Source_string identifies the string to be searched for the number of occurrences of *search_string*. The *source_string* may be a literal string, constant, or variable. In the following example:

```
ASTERISKS = COUNT ("A*BB*CCC*DDDD*E", "*")
```

the value of 4 will be assigned to the variable identifier ASTERISKS because * (asterisk) occurs four times.

Each character in *search_string* is matched to the *source_string* only once. Thus in the following example:

```
BBS = COUNT ("BBBBBB", "BB")
```

the value of 5 is assigned to the variable BBS. The first character is the beginning of the first match, the second character is the beginning of the second match, and so on.

search_string

Search_string identifies the string to search for in *source_string*. The *search_string* may be a literal string, constant, or variable.

If *search_string* does not appear in the *source_string*, a 0 (zero) will be returned.

If *search_string* is null, every character in the *source_string* will be counted.

Correct Use of COUNT Function

```
X = "ABRACADABRA"
Y = "AB"
Z = "BR"
A = COUNT (X, Y)
B = COUNT (X, Z)
C = COUNT (X, "A")
D = COUNT (X, "")
```

The variable identifiers will be assigned as follows: A = 2, B = 2, C = 5, D = 11

```
R = "CK1/CK2/CK3/CK4"
SM = "/"
N = COUNT (R, SM)
```

N will be assigned the value 3.

Incorrect Use of COUNT Function

```
COUNT (A)
```

One of the two required expressions is missing; there is a space between the function and parenthesis (not allowed)

```
W = "ABRACADABRA"
X = COUNT (W, CA)
```

Because the literal string CA is not enclosed in quotation marks, the compiler will consider it a variable identifier.

DATA

The DATA statement sets up an area for data values that are to be read into the program by subsequent INPUT statements.

```
DATA expression [{, expression} ::]
```

expression

An *expression* may be any valid combination of variables, functions, or literals. If more than one *expression* is specified, the *expressions* are delimited by commas.

- ☞ If a variable is used, it must not contain a record mark (ASCII character 255).

The *expressions* are processed in order. The INPUT operation will process each subsequent *expression* until all the *expressions* have been exhausted.

At any further request for data from an INPUT statement, a prompt will request input of additional data from the user. The DATA statement loads the @DATA variable using ASCII character 13 to separate the result of each *expression*.

- ☞ For more information, see CLEARDATA and PROMPT.

Correct Use of DATA

```
DATA "CST", "288", "1", "SMART-COMM", "JOHNSEN"
PERFORM "UPDATE"
END
```

```
PRINT @(-1)
* pgm update
PRINT "File name = ":
INPUT FILE.NAME
PRINT "Key = ":
INPUT KEY
OPEN FILE.NAME TO FILE ELSE CLEARDATA ; STOP "No file"
READ REC FROM FILE, KEY ELSE CLEARDATA;STOP "No rec"
PRINT "Field = ":
INPUT FIELD
PRINT "Old = ":
INPUT OLD
PRINT "New = ":
INPUT NEW
LOCATE OLD IN REC USING @VM SETTING POS ELSE
  CLEARDATA;STOP "INCORRECT DATA"
END
REC<FIELD, POS> = NEW
```

```
PRINT "CHANGED"
END
```

The first program loads the @DATA variable and performs the VOC record UPDATE. VOC record UPDATE line one is TCL and line two is RUN BP UPDATE. The VOC record UPDATE runs the program UPDATE that locates the old data in a multivalued field and replaces it with the new data. Each field of the DATA statement is loaded into each successive INPUT statement:

```
FILE.NAME = CST
KEY = 288
FIELD = 1
OLD = SMART-COMM
NEW = JOHNSEN
```

Incorrect Use of DATA

```
DATA A, B, C, D
E, F, G
Continuing lines are used.
```

DATE()

The DATE function returns the current internal system date.

```
DATE ( )
```

The Internal System Date

The internal system date is the number of days since December 31, 1967. This date has been designated internally as day 0 (zero), so all dates after that are represented as the number of days that have elapsed since that specific date. All dates previous to the day 0 (zero) are designated as the negative number of days prior to the reference date. For example, consider the following dates:

Date	Internal Representation
December 1, 1967	-30
January 5, 1968	5
July 1, 1979	4200
January 1, 2000	11689

For example, the following statement assigns the current string value of the internal date to the variable identifier B:

```
B = DATE ( )
```

To convert internal system date to its month, day, and year value, see OCONV and TIMEDATE().

Correct Use of DATE() Function

```
TODAY = DATE( )
```

Assigns the value of the current internal date to the variable TODAY.

```
WEEK.AGO = DATE( ) -7
```

Assigns to the variable identifier WEEK.AGO the current internal date minus 7 days.

Incorrect Use of DATE() Function

```
C = DATE(3)
```

A character is used in the parentheses. (The parentheses cannot contain any character.)

```
DATE( )
```

DATE() is used alone in the statement. (A function cannot be used alone in a statement.)

DEBUG

The DEBUG statement activates the interactive Debugger.

DEBUG

Using DEBUG

When the DEBUG statement executes, the following message appears, indicating that the program named is now under the control of the Debugger:

```
Line n 'file program' broke because the break key was hit.
```

The Debugger uses an exclamation mark (!) prompt. Type HELP to display a message containing a summary of the debugging commands.

- ☛ For more information about the Debugger, and for a listing of the Debugger commands, see “The Debugger” in the chapter “Programming with R/BASIC”.

Correct Use of DEBUG

```
DEBUG
```

Calls the interactive Debugger.

```
IF ERR.FLAG THEN DEBUG
```

Calls the Debugger if the ERR.FLAG has been set.

Incorrect Use of DEBUG

```
DEBUG Line 3
```

Uses an expression where none is allowed.

DECLARE

The DECLARE statement allows the use of cataloged, user-defined functions and subroutines in R/BASIC programs. Once declared, the function or subroutine may be referenced at any point in the program. The functions or subroutines must be declared before they are referenced.

```
DECLARE [FUNCTION | SUBROUTINE] name [ {, name} ::]
```

FUNCTION and SUBROUTINE

The FUNCTION and SUBROUTINE keywords identify *name* as the name of a function or subroutine.

name

The name(s) of a function or subroutine. Multiple functions or subroutines may be declared by placing a comma between the names.

- For more information, see CALL, COMMON, FUNCTION, RETURN and SUBROUTINE. See also “External Functions and Subroutines” in the chapter “Programming with R/BASIC”.

Correct Use of DECLARE

```
DECLARE FUNCTION COMP
INPUT INT.RATE
INPUT PERIODS
INPUT SUBTOT
TOTAL.COST = SUBTOT*COMP (INT.RATE, PERIODS)
PRINT TOTAL.COST
END
```

```
FUNCTION COMP (INT.RATE, PERIODS)
RATE = (1 + (INT.RATE / 100))**PERIODS
RETURN RATE
END
```

The program uses the declared function COMP to calculate the compound interest and add it to SUBTOT. The result assigned to TOTAL.COST is printed. COMP has been cataloged.

Incorrect Use of DECLARE

```
DECLARE CHECKER
CHECKER has not been declared as a subroutine or a function.
```

DELETE

The DELETE statement deletes a record from a file.

```
DELETE [filevar  
      [CURSOR cursorvar]] ,key THEN|ELSE statements
```

filevar

Filevar is a file variable created by a previous OPEN statement.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable and is preceded with the keyword CURSOR. Cursor variables are initialized with a SELECT ... BY statement.

- ✎ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

key

The record referenced by *key* will be deleted from the file identified by *filevar* or the file accessed using the cursor in *cursorvar*.

THEN

The statement(s) following THEN are executed if a record is deleted successfully.

- ✎ For more information about the use of THEN and ELSE, see IF.

ELSE

The statment(s) following ELSE are executed if the record in *variable* cannot be deleted. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming With R/BASIC”.

- ✎ For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in a DELETE statement are optional. However, it is strongly recommended that all DELETE logic be constructed to accommodate possible unsuccessful deletes. If no THEN or ELSE clause is specified, any errors other than attempting to delete a non-existent record are passed to the system error handling routine.

Correct Use of DELETE

```

DECLARE SUBROUTINE MSG, FSMSG
OPEN "CUSTOMER" TO FILE_CUST THEN
  FLAG = ""
  SELECT FILE_CUST
  LOOP UNTIL FLAG = "STOP"
    READNEXT ID THEN
      READ INFO FROM FILE_CUST, ID THEN
        RESP = "N"
        TEXT = "Delete ":ID"?":
        TEXT<2>= "Are you sure (Y/N)?":
        MSG(TEXT,"RC",RESP,"")
        IF RESP[1,1]="Y" THEN
          DELETE FILE_CUST, ID ELSE
            FSMSG()
          END ; * delete
        END ; * if resp[1,1]
      END ; * read
    END ELSE
      FLAG = "STOP"
    END ; * readnext
  REPEAT
END ; * open

```

The program selects the file. As each record is read, it prints the key and the first field. If a Y is entered the record is deleted.

Incorrect Use of DELETE

```
DELETE FILE.CM,
```

The key is missing.

```
DELETE *
```

The parameters are not specified.

DELETE()

The DELETE function is used to delete the data contents of a specified field, value, or sub-value from a dynamic array. The corresponding delimiter is also deleted.

```
DELETE (array, field, value, subvalue)
```

array

Array is the dynamic array that is to be searched.

- ☞ For information on dynamic arrays, see “Dynamic Arrays” in the chapter “Programming with R/BASIC”.

field

Field is the field number of the field to be deleted. The *value* and *subvalue* values must be 0 (zero) to delete *field*.

value

Value is the value number of the value to be deleted. The *subvalue* value must be 0 (zero) to delete *value*.

subvalue

Subvalue is the subvalue number of the subvalue to be deleted. The values of *field* and *value* do not affect the deletion of *subvalue*.

☞ For more information, see EXTRACT, REPLACE, and INSERT.

Correct Use of DELETE Function

```

DECLARE SUBROUTINE MSG
OPEN 'CST' TO FILE.CST THEN
  SELECT FILE.CST
  DONE = 0
  LOOP
    READNEXT @ID ELSE DONE = 1
  UNTIL DONE DO
    READ @RECORD FROM FILE.CST, @ID THEN
      NEW.RECORD = DELETE(@RECORD,4,0,0)
      WRITE NEW.RECORD ON FILE.CST, @ID ELSE
        MSG("Unable to write %1% on CST file!", "", "", @ID)
      END ; * write
    END ; * read
  REPEAT
END ELSE
  MSG("Unable to open CST file!", "", "", "")
END

```

The program loops through the CST file and deletes the fourth field of each record.

```

* multivalued field with 4 names
NAMES = "KEN" : @VM : "JANET" : @VM : "NATHAN" : @VM : "SABRINA"
NAME.TO.DELETE = ""
CALL MSG("Delete which name?", "R", NAME.TO.DELETE, "")
LOCATE NAME.TO.DELETE IN NAMES USING @VM SETTING POS THEN
  NEW.NAMES = DELETE(NAMES, 1, POS, 0)
END ELSE
  CALL MSG("No such name!", "", "", "")
END
STOP

```

The program prompts the user to delete a name from an array of four names (separated with value marks). When the user has specified the name to delete, the program locates the name in the array, and deletes it. Note that the field argument of the DELETE function is set to 1, since the entire array is a single field.

Incorrect Use of DELETE Function

```
DELETE (Z,1,0,0) = "WRONG"
```

The function appears on left side of the assignment operator.

DIMENSION

The DIMENSION statement declares the name of a matrix and allocates the storage space the matrix will require. The DIMENSION statement must be used before the matrix can be referenced in a program or subroutine.

The synonym DIM may be used instead of DIMENSION.

```
DIMENSION matrix(row {,column}) [, matrix(row {,column}) ::]  
(or)  
DIM      matrix(row {,column}) [, matrix(row {,column}) ::]
```

Using DIMENSION

R/BASIC automatically allocates space for a 0 (zero) element when a matrix is dimensioned. That is, one more element is always available to receive data than is dimensioned in the DIMENSION statement. It is a single element, not a row or column. The data in the 0 (zero) element can be accessed by using a 0 (zero) subscript:

```
MATRIX(0)
```

The maximum number of elements in a matrix cannot be increased during the execution of a program. A one-dimensional matrix cannot be reassigned to be a two-dimensional matrix.

- ✎ The DIMENSION statement only names the matrix and defines its dimensions; it does not assign values to the elements.
- ✎ For more information, see COMMON, MAT, and “Matrices” in the chapter “Programming with R/BASIC”.

matrix

Any legal identifier may be used as the *matrix* name. If more than one *matrix* is declared, the full declarations (name, row, and optional column) are delimited by commas.

row

Row is an integer identifying the number of rows in *matrix*. Numbers resulting from expressions will be truncated to their integer value.

column

Column is an integer identifying the number of columns in *matrix*. Numbers resulting from expressions will be truncated to their integer value. The *row* and *column* arguments must be separated by a comma.

Correct Use of DIMENSION

```
DIM MONTH(12), YEAR(12, 5)
```

Two matrices are dimensioned. MONTH has 13 elements, YEAR has 61 elements (12 rows by 5 columns plus a 0 (zero) element).

```
DIM TESTB(X, Y)
```

The number of rows in matrix TESTB is the current value of X and the number of columns is the current value of Y.

```
DIM V(10), K(10), R(10)
```

Matrices named V, K, and R each to contain 11 elements.

Incorrect Use of DIMENSION

```
DIM W(5) X(7)
```

Comma between the matrices is missing.

```
DIM C(6, 0)
```

Reference is made to the 0 (zero) element and a row at the same time.

```
DIM(4, 5)
```

A matrix variable name is missing.

DIR()

DIR() returns directory information (size, date, time) for an operating system file.

DIR(filename)

Using DIR()

The DIR function returns the same information about an operating system file as is normally displayed in an operating system DIR command. This includes the size of the file, and the date and time the file was last updated.

DIR returns a three-element dynamic array of file information (@FM-delimited), using this layout:

- <1> file size in bytes
- <2> last update date (in Advanced Revelation internal format)
- <3> last update time (in Advanced Revelation internal format)

You can convert the internal forms of the date and time to external format using the OCONV function.

filename

The *filename* is the an expression yielding the full name (including path, if appropriate) of the operating system file.

- ☛ For more information about directories and pathnames, see your operating system manual. Related R/BASIC commands are DIRLIST() and INITDIR.

Correct Use of DIR()

```
FILE_INFO = DIR("C:\AUTOEXEC.BAT")
SIZE_ = FILE_INFO
DATE = OCONV(FILE_INFO, "D2-")
TIME = OCONV(FILE_INFO, "MTH")
TEXT = "File information for AUTOEXEC.BAT"
TEXT1 = "Size = ":SIZE
TEXT1 = "Date = ":DATE
TEXT1 = "Time = ":TIME
CALL MSG(TEXT, "", "", "")
STOP
```

The program displays file information about the AUTOEXEC.BAT file.

```
DECLARE SUBROUTINE MSG
* get information about path to list
PATH = DRIVE()
MSG("Path for listing", "R", PATH, "")
* add backslash if missing
IF PATH[-1,1] NE "\" THEN PATH := "\"

* get info about files to list
MASK = "*. *"
MSG("File mask", "R", MASK, "")

* establish files to list
INITDIR PATH:MASK

* get file names from directory
FILE_LIST = ""
DONE_ = 0
LOOP
  * fetch about 1K of file names until null returned
  FILES = DIRLIST()
  IF FILES EQ "" THEN DONE = 1
  FILE_LIST := FILES
UNTIL DONE REPEAT
```

```

HEADING "File listing for ":PATH:MASK:"'L'"

* list files
FILE_CNT = COUNT(FILE_LIST, @FM)
FOR FILE_CTR = 1 TO FILE_CNT

    NEXT_FILE = FILE_LIST< FILE_CTR >
    FILE_INFO = DIR( PATH : NEXT_FILE )

    SIZE = FILE_INFO<1>
    DATE = OCONV(FILE_INFO<2>, "D2-")
    TIME = OCONV(FILE_INFO<3>, "MTH")

    PRINT NEXT_FILE "L#14" :
    PRINT SIZE      "R#8" :
    PRINT DATE      "R#10" :
    PRINT TIME      "R#8"
NEXT FILE_CTR
PRINT
PRINT FILE_CNT : " file(s) listed."
STOP

```

The program provides a listing similar to that of an operating system DIR command.

DIRLIST()

The DIRLIST() function returns a group of operating system filenames from a directory indicated by the INITDIR statement.

DIRLIST()

Using DIRLIST()

DIRLIST() returns a dynamic array of the operating system filenames found in the directory masked by the INITDIR statement.

With each use of DIRLIST() about 1K of filenames will be returned from the indicated directory. DIRLIST() may be placed in a loop to obtain all the filenames in a large directory. The loop would continue until the list is empty. Each filename is terminated with a field mark (ASCII character 254).

- ✎ For more information about directories and pathnames, see your operating system manual.

Correct Use of DIRLIST()

```

DRIVE = "C:\HISTORY\DOCS\LETTERS\"
INITDIR DRIVE: "*.DOC"
LOOP
  LIST = DIRLIST()
UNTIL LIST = ""
  PRINT LIST
  INPUT X
  COL = 0
  LOOP
    REMOVE FILE.NAME FROM LIST AT COL SETTING FLAG
  UNTIL FLAG = 0 DO
    PRINT DRIVE: FILE.NAME
  REPEAT
REPEAT
END

```

The program gets the list of files in the C:\HISTORY\DOCS\LETTERS\ directory and prints them. It uses INITDIR to indicate the directory and DIRLIST() to return the filenames.

Incorrect Use of the DIRLIST() Function

```
LIST = DIRLIST(DRIVE)
```

A variable is placed in the parentheses. (The parentheses cannot contain any variable.)

DRIVE()

The DRIVE() function returns the default drive.

DRIVE ()

Using DRIVE()

The value returned will be the current operating system path for the default drive, directory, subdirectory, or volume. The default drive is the drive current at Advanced Revelation logon time.

Correct Use of DRIVE()

```

PRINT "Drive " : DRIVE() : "is current"
IF DRIVE() EQ "T:" THEN STOP

```

This program prints the current drive letter. If the drive letter is "T:", the program stops.

Incorrect Use of DRIVE()

```
DRIVE(2)
```

A character is used in the parentheses. (The parentheses cannot contain any character.)

ECHO

The ECHO statement either suppresses or enables display of input characters on the terminal screen.

```
ECHO [ON
      OFF
      expression]
```

ON

This is the default setting. Any character input displayed on the screen.

OFF

The input characters are not to be displayed on the terminal screen. This is useful for programs that use keyboard characters for cursor control or game input.

expression

If the *expression* evaluates to true (non-zero), ECHO ON is set. If the value of the *expression* is false (zero) or null, ECHO OFF is set.

The PROMPT command may be used to suppress the input prompt.

☞ For more information, see GETECHO, INPUT and PROMPT.

Correct Use of ECHO

```
ECHO OFF
INPUT PASSWORD, 8
```

The characters that are typed at INPUT will not appear on the terminal screen.

```
ECHO ON
```

Characters will appear on the terminal screen.

```
IF FLAG THEN ECHO OFF
```

Turns ECHO off if flag is set.

Incorrect Use of ECHO

```
ECHO
```

ON or OFF is missing in the statement.

```
ON ECHO GOTO 1
```

Invalid syntax is used.

END

The END statement can identify the end of a block of statements, the physical end of a program, or it can terminate the program.

```
END
```

Using END

Use END to designate the physical end of alternative sequences within such statements as IF, READ, and OPEN. If a program does not contain an END statement for each logical block of statements, the effect upon program execution will be undetermined.

When used as the last statement in R/BASIC, END also terminates the program. Comment statements may be added after the last END statement in a program.

Program compilation will stop at the first END that is not part of a multiline statement. Therefore, use STOP instead of END to terminate the execution of a program.

☞ For more information, see IF.

Correct Use of END

```
IF B > 500 THEN
  PRINT "YOU WIN"
END
```

If the current value of B is more than 500, YOU WIN is printed.

```
READ X FROM Y ELSE
  PRINT 'NOT THERE'
STOP
END
```

If the record X does not exist in file Y, the literal string "NOT THERE" will be printed and the program will stop.

Incorrect Use of END

```
END (SUBROUTINE)
```

Uses a parameter where none is allowed.

```
END IF N = 1
```

The conditional statement will not be executed.

EQUATE

The EQUATE statement allows the definition of a symbol as the equivalent of another variable or constant. The symbol may be equated to a variable, character, character string, number, or function, and may be used interchangeably with its equated value throughout the program.

Only one EQUATE statement may appear on a line.

```
EQUATE symbol TO expression
      (or)
EQU    symbol TO expression
```

symbol

During compilation, every occurrence of *symbol* in the program will be replaced with the specified *expression*. Though *symbol* is treated as the equivalent of the *expression*, it is not allocated storage in memory.

expression

The *expression* may be any legal R/BASIC expression.

Correct Use of EQUATE

```
EQUATE TRUE TO 1
EQUATE FALSE TO 0
```

The symbols TRUE and FALSE are compiled as the equivalent of 1 (one) and 0 (zero), respectively.

```
EQU EM$NAME TO EM.REC(5)
```

EM\$NAME equates to the fifth element in array EM.REC.

```
EQU CANT TO "CAN'T OPEN FILE"
```

CANT will equate to the literal string "CAN'T OPEN FILE".

```
EQU TAX TO TOTAL * .085
EQU DISCOUNT TO TOTAL *.1
LOOP
```

```
    PRINT 'ENTER AMOUNT ' :
    INPUT AMOUNT
UNTIL AMOUNT = 'END'
    TOTAL += AMOUNT
```

```
REPEAT
PRINT 'TOTAL AMOUNT ' : TOTAL : 'SALES TAX ' : TAX
PRINT 'YOU MAY DEDUCT 10% OR ' : DISCOUNT
PRINT 'IF PAID BY THE 10TH'
END
```

The equated symbols TAX and DISCOUNT are resolved when they are printed out.

Incorrect Use of EQUATE

EQUATE 3.1416 TO PI

The symbol should have appeared first.

EQUATE 10 TO TEN

Uses a number for the symbol.

EXECUTE

The EXECUTE statement sends a command to The Command Level (TCL) for execution and then returns to the calling program.

EXECUTE <i>command</i>

command

The *command* must be a valid TCL command, a cataloged program, or null. Null will move to the next TCL level. After EXECUTE makes a recursive call to the TCL window, control proceeds to the statement following the EXECUTE statement.

Use EXECUTE to suspend all current processing to execute another subsystem or process. Unlike PERFORM, the following information is saved and not passed to the new level of TCL:

- The currently active list of keys
- The contents of @DATA assigned by the most recent DATA statement
- The user defined variables @RECUR0 - @RECUR4
- The printer status (the printer is turned off.)
- The BREAK status (BREAK is turned on.)
- The ECHO status (ECHO is turned on.)

Use PERFORM when you wish to pass the above information between levels.

The number of levels that may be recursively executed is limited to 9.

- ☞ For more information, see CALL and PERFORM. See also "Special R/BASIC Variables" in the chapter "Programming with R/BASIC".

Correct Use of EXECUTE

EXECUTE ""

Moves to the next recursive level.

EXECUTE "ACCOUNTS.PAYABLE"

Invokes the ACCOUNTS.PAYABLE VOC item.

```
CMD = "LIST ONLY DOS"
EXECUTE CMD
```

Produces a list of DOS filenames.

Incorrect Use of EXECUTE

```
EXECUTE
```

Expression is missing.

```
EXECUTE "SELECT CUSTOMER"
```

Select list will be lost on return.

```
EXECUTE EDIT INVOICES A101
```

Quotes on the command string are missing.

EXP

The EXP (exponential) function calculates the result of base *e* raised to the power designated by the value of the expression. It is the inverse of the LN function.

EXP (expression)

expression

The *expression* must evaluate to a numeric value. EXP calculates the value of base *e* raised to the power designated in the *expression*.

The value returned by EXP is calculated according to the following formula:

$$\text{EXP function value} = (\text{base } e)^{(\text{expression})}$$

☛ For more information, see LN.

Correct Use of EXP Function

```
PRINT 'EXPRESSION', 'EXP (EXPRESSION)'
FOR EXPRESSION = 0 TO 4
  PRINT EXPRESSION, EXP (EXPRESSION)
NEXT
END
```

This program will generate the following table:

EXPRESSION	EXP(EXPRESSION)
0	1
1	2.71828
2	7.389
3	20.0855
4	54.5981

Incorrect Use of EXP Function

Y = EXP 4
Parentheses are missing.

EXPENDABLE

The EXPENDABLE statement is used with the keywords SUBROUTINE or FUNCTION to indicate that the routine is to be removed from memory as soon as it terminates.

```
EXPENDABLE [ SUBROUTINE ] name { (argument { , argument :: ) } }  
           [ FUNCTION   ]
```

Using EXPENDABLE

EXPENDABLE tags a subroutine or function as one that can be removed from memory as soon as it terminates execution.

When a subroutine or function is called, its object code remains in memory until the main program that called it executes a STOP statement. Because subroutines remain resident, subsequent calls to the subroutine do not need to read the object code from disk. However, a main program that calls many subroutines may significantly reduce the amount of available memory.

A subroutine designated as expendable will be flushed from memory as soon as it finishes execution. This releases the memory used by the subroutine back to the pool of string space.

If the same subroutine is called again, it must be re-read from disk. Routines tagged as expendable are therefore typically ones that will be called infrequently.

name

The name of the subroutine or function to be declared expendable.

argument

The argument(s) required by the subroutine or function. The *argument* is enclosed in parentheses. Multiple arguments are delimited with commas.

For more information on subroutines and functions, see the commands **SUBROUTINE** and **FUNCTION**, and “Subroutines” in the chapter “Programming with R/BASIC”.

Correct Use of EXPENDABLE

```
* main program
EXIT = 0
LOOP
  GOSUB DISPLAY_MENU
  GOSUB GET_CHOICE
  WHILE NOT(EXIT) DO
    BEGIN CASE
      CASE CHOICE = 1
        CALL DO_REPORT
      CASE CHOICE = 2
        CALL SHOW_WINDOW
      CASE CHOICE = 3
        EXIT = 1
    END CASE
  REPEAT
```

```
EXPENDABLE SUBROUTINE DO_REPORT
* this is the (separate) expendable subroutine
COMMAND = "LIST CUSTOMER BY ST COMPANY CITY ST"
PERFORM COMMAND
RETURN
```

The main program displays a menu and gets user input. If the user enters 1, the program calls the expendable subroutine DO_REPORT. Upon termination of DO_REPORT, the subroutine is removed from memory, and control returns to the main program. Note: other subroutines referenced in the example are not shown.

EXTRACT

The **EXTRACT** function extracts a specified field, value, or subvalue from a dynamic array.

EXTRACT(array, field, value, subvalue)

array

The *array* specifies the dynamic array that is to be accessed.

field

The *field* argument is the field number of the field in *array* to extract. The *value* and *subvalue* values must be 0 (zero) to extract the *field*.

value

Value is the value number of the value to be deleted. The *subvalue* value must be 0 (zero) to delete *value*.

subvalue

Subvalue is the subvalue number of the subvalue to be deleted. The values of *field* and *value* do not affect the extraction of *subvalue*.

Compiled code will run more efficiently when *field* is an integer constant and *value* and *subvalue* are equal to zero.

☞ For more information, see < > (Angle Bracket Operators), DELETE, INSERT, and REPLACE.

Correct Use of EXTRACT Function

```
IF EXTRACT(Y, 3, 2, 1) > 6 THEN
  STOP
END
```

If subvalue 1, of value 2, of field 3, of dynamic array Y is greater than 6, the program will stop.

```
FOR I = 1 TO 31
  T = EXTRACT(CASH, I, 0, 0)
  PRINT T
NEXT I
```

Extracts 1 to 31 fields from dynamic array CASH and prints the values.

Incorrect Use of EXTRACT Function

```
X = EXTRACT(D, 2, 4)
```

One expression is missing.

```
X = EXTRACT (D, 2, 4, 2)
```

Uses space between EXTRACT and parentheses where no space is allowed.

FIELD

The FIELD function returns a substring from a string expression. The substring is specified by a delimiter character and index position.

FIELD(string, delimiter, occurrence{, field})

string

String is the source string for the FIELD operation.

delimiter

The *delimiter* may be any type of data, including field, value or subvalue characters, or null. If more than one character occurs in the delimiter expression, only the first character is used.

If the *delimiter* does not exist in *string*, one of two possible values will be returned:

- If the first occurrence of *delimiter* is specified, the entire *string* is returned.
- If another occurrence is specified and either the *delimiter* or the occurrence does not exist, a null value is returned.

occurrence

The *occurrence* defines the number of the occurrence of the substring that is to be returned. If the occurrence *delimiter* evaluates to less than 1 (one), 1 (one) is assumed.

field

The *field* parameter defines the number of successive fields after the occurrence of *delimiter* that are to be returned with the substring. The delimiter(s) will be included as part of the returned value. A 1 (one) is assumed if field is null or less than 1 (one).

Also see [] (bracket), COL1() and COL2().

Correct Use of FIELD Function

```
A = "32#33#34#35"
B = "#" ; C = 33
IF FIELD(A,B,2) = C THEN STOP
END
```

The program segment will terminate because the value returned by the FIELD function is equal to C, making the test condition true.

```
VAR = "$$$A!!!A; ; ;"
S = FIELD(VAR,"A",2)
PRINT @(5,1) :S
```

The variable identifier S will be assigned the value !!! and printed at column 5 line 1.

Incorrect Use of FIELD Function

```
X = FIELD("R*DS*GTh", "*")
```

The occurrence expression is missing.

FIELDSTORE

The FIELDSTORE function is used to replace, delete, or insert substrings in a specified string. The substrings are identified by a specified delimiter character.

FIELDSTORE (**source_string**, **delimiter**, **occurrence**, **field**, **string**)

source_string

The *source_string* specifies the string that is to be searched.

delimiter

The *delimiter* is any single ASCII character including field, value, or subvalue marks.

occurrence

Processing starts at the substring designated by *occurrence*. If the number of substrings within *source_string* is less than the *occurrence*, the *source_string* will be padded with the necessary number of null substrings before FIELDSTORE is executed.

field

The *field* specifies the number of substrings in string to apply to *source_string*. The value of *field* determines the operation:

- If *field* is a positive number, that number of substrings will be moved from the *string* to *source_string*.
- If *field* is a negative number, the entire *string* is inserted in *source_string*. The *field* specifies the number of substrings to replace in *source_string*.
- If *field* is 0 (zero), the entire *string* is inserted in the *source_string*. The insertion occurs at the substring indicated by *occurrence*.

string

The *string* contains the new string values to store in *source_string*.

Correct Use of FIELDSTORE

```
X = "206"  
X = FIELDSTORE (X, '-', 2, 1, "226")  
X = FIELDSTORE (X, '-', 3, 1, "9362")  
PRINT X
```

This would print 206-226-9362.

```

PROMPT ""
LOOP
  PRINT
  PRINT "INPUT STRING # 1 " :
  INPUT STRING1
  WHILE STRING1 NE 'END'
    PRINT "INPUT DELIMITER " :
    INPUT DELIM
    PRINT "INPUT OCCURRENCE " :
    INPUT OCCUR
    PRINT "INPUT FIELD " :
    INPUT FIELD
    PRINT "INPUT STRING #2 " :
    INPUT STRING2
    AFTER = FIELDSTORE (STRING1, DELIM, OCCUR, FIELD, STRING2)
    PRINT "BEFORE = " : STRING1
    PRINT "AFTER = " : AFTER
  REPEAT

```

The program prints the input string before and after the FIELDSTORE function.

Incorrect Use of FIELDSTORE Function

```
Z = FIELDSTORE (B, C, R)
```

Uses only three parameters where five are required.

FLUSH

The FLUSH statement clears inactive file buffers and recovers available space in random access memory.

Use FLUSH with the GARBAGECOLLECT statement.

FLUSH

Using FLUSH

Some of the file types available to Advanced Revelation will buffer or cache data. Groups of records are sent to a temporary buffer when one of the records in the group is accessed. The group of records then stays in the buffer until the space is needed for another group of records. If the space is not needed, the original group will remain in the buffer. The FLUSH statement will clear out any of the groups that are not currently being accessed.

FLUSH also writes to disk any of the buffers that contain records that have been changed. As a result, all unused memory is then available to the program.

☞ For more information, see GARBAGECOLLECT.

Correct Use of FLUSH

FLUSH
GARBAGECOLLECT
These two statements will recover unused space in memory and flush any changed buffers to disk.

Incorrect Use of FLUSH

FLUSH BUFFER
Expression is not allowed.

FMT

The FMT function changes data to a specified formatted pattern. This pattern may include field width, background fill characters, line justification, conversion specifications, and masking.

FMT(string, format)

string

The *string* is any character string to format on output.

format

The *format* argument may be one of the following parameters:

justification{ (*fill_char*) }*mask*
justification{ (*fill_char*) }#{*field_size*}
conversion

justification	All <i>formats</i> must specify one of the following justifications:	
	L	Left justification
	R	Right justification
	C	Center justification
	T	Text justification: left justify and insert text marks (ASCII character 251) as line delimiters.

- field_size** The optional *field_size* specifies the size of the field in which the value is to be justified. It must yield an integer. If the output data has a length larger than *field_size*, it will be truncated.
- mask** The *mask* is the template of the output that is desired. For example, ###-##-#### would be the template or mask that could be used for a variable that will be output as a Social Security number. The # characters are replaced with the pertinent numbers from the variable while the - characters are placed into the output according to the mask pattern.
- fill_char** The *fill_char* is a single character enclosed by parentheses that replaces the leading or trailing blank spaces. The default fill character is a space.
- conversion** Standard output conversions can be specified. For more information, see OCONV.

Correct Use of FMT Function

Example	Output
FMT ("12345", "L#6")	"12345 "
FMT ("12345", "R(*)#8")	"***12345"
FMT ("ABCDEFG", "R#4")	"DEFG"
FMT ('ABCD', 'C#6')	" ABCD "
FMT (6666, "D2-")	"04-01-86"
A = FMT ("1234567890", "L(###)###-####")	" (123) 456-7890"

```
PRINT @(-1); *clear screen
LOOP
  PRINT "Enter string to format:  " :
  INPUT STRING
UNTIL STRING = 'END' DO
  PRINT "Enter the format:  " :
  INPUT FORMAT
  OUT1 = FMT (STRING, FORMAT)
  PRINT OUT1
  OUT2 = 'FORMAT': '=FMT ("':STRING:',' ", "'':FORMAT: '")'
  PRINT OUT2
REPEAT
END
```

The program will format the input string using the input format.

Incorrect Use of FMT Function

```
PRINT X 'L##-#5'
```

Mixes mask pattern with format pattern.

FOOTING

The FOOTING statement prints or displays a character string at the bottom of each page at the printer or the terminal.

```
FOOTING [string{'options'}::]
```

string

The *string* parameter must yield a legal R/LIST FOOTING string. The *string* is enclosed in quotes.

☞ See “Format Keywords” in the “R/LIST Keyword Reference” chapter of *R/LIST*.

options

The *options* are included within the double quotes enclosing the *string*. Each option must be enclosed in single quotes. Valid options for FOOTING are:

Option	Meaning
'B'	Inserts the value of the R/LIST BREAK-ON setting into the footing. R/BASIC uses the value of @BREAK.
'D'	Inserts the current date as DD MM YYYY
'F'	Inserts the current file being read
'L'	Causes a line feed. 'LL' will leave one line blank.
'N'	Suppresses automatic paging
'P'	Inserts current page number
'PP'	Inserts current page number with right justification in a four-space field
'T'	Inserts the current time and date
' '	Prints a single quote in the footing

☞ For more information, see HEADING.

Correct Use of FOOTING

```

CLEAR
PRINT "INPUT STARTING DATE  " : ; INPUT START.DATE
PRINT "INPUT ENDING DATE  " : ; INPUT END.DATE
FOOT = " Report for Period Starting  "
FOOT := START.DATE : " Thru  " : END.DATE
FOOTING FOOT
OPEN 'INV' TO FILE.INV THEN
PERF = "SELECT INV WITH INV.DATE BETWEEN  "
PERF := QUOTE(START.DATE) : " AND " : QUOTE(END.DATE)
PERFORM PERF
  LOOP UNTIL FLAG = "STOP"
    READNEXT @ID THEN
      READ @RECORD FROM FILE.INV,@ID THEN
        PRINT @ID, @RECORD<1>, @RECORD<2>
      END ELSE STOP
    END ELSE
      FLAG = "STOP"
    END
  REPEAT
END

```

The program provides a listing the key and fields 1 and 2 from the INV file. The footing is created dynamically based on user input.

Incorrect Use of FOOTING

```

FOOTING Daily Report 'PD'

```

Missing double quotes.

FOR ... NEXT

The FOR and NEXT statements specify the beginning and ending points of a program loop. This loop is an operation or sequence of operations repetitively executed under the control of an index variable designated in the FOR statement.

```

FOR variable = start TO end {STEP amount}
  {statements}
  {[WHILE] conditions {statements}}
  [UNTIL]
NEXT {variable}

```

Using FOR ... NEXT

Each FOR can have only one NEXT statement, although FOR/NEXT loops may be nested any number of times. Each NEXT is assumed to be associated with the closest preceding FOR that has not been paired with a NEXT. The *variable* reference following NEXT is optional, but recommended.

The NEXT statement indicates the ending point of the loop. Control then passes to the FOR statement, where the value of *variable* is incremented and the termination test is repeated.

If the loop does terminate, control is transferred to the statement following the NEXT statement.

The *variable* may be referenced within a loop. However, it is not good programming practice to change the *variable* value. Resetting the *variable* to end the loop is an exception.

When the FOR/NEXT conditions have been satisfied, the value of the *variable* represents the number of times the loop was executed. If the loop is not executed, the initial value of *variable* is unchanged.

If a GOTO or GOSUB is used to pass control into an inactive FOR/NEXT loop, unpredictable results may occur.

Before entering a loop, execute any expressions within the loop that do not change value. This improves efficiency, as does keeping the FOR/NEXT statement as simple as possible.

variable

The *variable*, also called the loop variable or index variable, is assigned the current iteration value.

The value of *variable* changes, beginning with the *start* value, and incrementing by the specified amount as each successive loop is executed, until the *end* value is reached.

start

Start is an integer to assign as the beginning value of *variable*.

end

End is an integer indicating the maximum value of *variable*.

The value of *variable* is tested against the *end* value at the beginning of each successive loop. If the test does not cause termination, the statements between the FOR and NEXT statements will be executed. When the *end* value is exceeded, the loop terminates.

STEP

The STEP keyword identifies the value following as the amount to increment or decrement *variable* each time the loop is executed. If STEP is omitted, the default increment is 1.

☞ *Amount* can be either a positive or a negative integer value.

WHILE

The WHILE keyword limits or controls the iteration of the FOR ... NEXT loop. *Conditions* can be any valid R/BASIC expression(s). If the *conditions* evaluate to true (non-zero), the FOR ... NEXT loop is executed. If the *conditions* are false (0), the FOR ... NEXT loop terminates and program control passes to the statement immediately following the NEXT statement.

UNTIL

The UNTIL keyword limits or controls the iteration of the FOR ... NEXT loop. If the *conditions* evaluate to false (0), the FOR ... NEXT loop is executed. If the *conditions* are true (non-zero), the FOR ... NEXT loop terminates and program control passes to the statement immediately following the NEXT statement.

Correct Use of FOR ... NEXT

```
FOR X = 5 TO 50 STEP 5
  PRINT X
NEXT X
```

The loop will execute 10 times. Numbers 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 will be printed.

```
* FOR/NEXT WITH UNTIL
TOTAL=0
FOR Z = 1 TO 10
  PRINT 'ENTER VALUE ':Z:
  INPUT TEMP.AMT
  UNTIL TEMP.AMT = ''
    TOTAL += TEMP.AMT
  NEXT Z
PRINT 'TOTAL IS ': TOTAL
PRINT 'AVERAGE IS ': TOTAL/(Z-1)
```

The program will continue until you press [Enter] at the prompt. or until 10 values have been collected. The total and the average will be printed.

Incorrect Use of FOR ... NEXT

```
FOR A = 10 WHILE C = 0
```

TO expression is missing.

```
FOR A = 1 TO 10 UNTIL B = 25 STEP 4
```

STEP expression is used before UNTIL clause.

FUNCTION

A FUNCTION statement identifies a program as an external function. It must be the first line in the function. It is used to specify the name and arguments that other programs must use when declaring the function.

```

FUNCTION function {argument [ {, argument} ::]
    statements
RETURN expression

```

function

Any legal variable identifier may be used as the name of the *function*.

The *function* must have been previously compiled and cataloged before it can be called.

The *function* must have logic that calculates the desired result as an expression or a variable. This result must be included in a RETURN statement located at a logical termination of the program.

For more information, see DECLARE, RETURN, and EXPENDABLE. See also "External Functions and Subroutines" in the chapter "Programming with R/BASIC".

Once a *function* has been called, it remains in memory until the main program that called it executes a STOP statement, or until a RESET or debugger END command is executed. For information on removing subroutines from memory, see EXPENDABLE.

argument

The *argument* is optional and can contain a variable, matrix, constant, and expression. If used, the argument must be enclosed in parentheses. If no arguments are used, the parentheses are omitted. Multiple arguments are delimited with commas.

The arguments can be used to pass values to and from the calling declared program and the function. The calling declared program must use the same number of arguments as are specified in the function to be called.

Correct Use of FUNCTION

```

DECLARE FUNCTION COMP
PRINT "Please enter the interest rate:":
INPUT INT.RATE
PRINT "Please enter the number of periods:":
INPUT PERIODS
PRINT "Please enter the loan total:":
INPUT SUBTOT
TOTAL.COST=SUBTOT*COMP (INT.RATE,PERIODS)
PRINT TOTAL.COST
END

```

Function Program

```
FUNCTION COMP (INT.RATE, PERIODS)
RATE = (1 + (INT.RATE/100))**PERIODS
RETURN RATE
```

The program uses the declared function COMP to calculate the compound interest and add it to SUBTOT. The result, TOTAL.COST, is printed.

Incorrect Use of FUNCTION

```
FUNCTION
READ RATE FROM CODE.FILE, RESULT ELSE
  ERROR = RESULT : " Bad Key "
  RETURN 1
END
RETURN RATE
```

The name of the FUNCTION is not specified.

GARBAGECOLLECT

The GARBAGECOLLECT statement forces a recovery of all unused or abandoned string space.

GARBAGECOLLECT

String Space

R/BASIC constantly recovers string space at regulated intervals. GARBAGECOLLECT forces recovery of the available string space between these intervals.

☞ GARBAGECOLLECT is ordinarily used only after a FLUSH statement.

Correct Use of GARBAGECOLLECT

```
GARBAGECOLLECT
```

Recovers all available string space

Incorrect Use of GARBAGECOLLECT

```
GARBAGECOLLECT 'X'
```

A parameter is used where none is allowed.

GETBREAK, GETECHO, GETPRINTER, GETPROMPT

The GET functions return the current settings for BREAK, ECHO, PRINTER, and PROMPT system variables.

```
GETBREAK ( )
GETECHO ( )
GETPRINTER ( )
GETPROMPT ( )
```

GETBREAK()

GETBREAK() returns the break value that is set by BREAK. BREAK ON sets the system variable to 1 (one). BREAK OFF sets the variable to 0 (zero).

GETECHO()

GETECHO() returns the value set by ECHO. ECHO ON sets the value to 1 (one) and ECHO OFF sets the value to 0 (zero).

GETPRINTER()

GETPRINTER() returns the value set by PRINTER. PRINTER ON sets the value to 1 (one) and PRINTER OFF sets the value to 0 (zero).

GETPROMPT()

GETPROMPT() returns the character set by PROMPT. PROMPT defines the character output by an INPUT statement.

For more information, see BREAK, ECHO, INPUT, PRINTER, and PROMPT.

Correct Use of GET Functions

```
PRINT "Send report to printer ":
INPUT ANS
IF ANS[1,1] = "Y" THEN PRINTER ON
P.OPT = GETPRINTER()
PRINTER OFF
CALL WRAPUP.SUB
IF P.OPT THEN
  PRINTER ON
  PRINT "Turn on the printer before running the report"
END
```

The program segment sets P.OPT and calls WRAPUP.SUB. P.OPT is tested after the subroutine to set the PRINTER variable.

Incorrect Use of GET Functions

```
GETECHO ( )
GETECHO( ) is not a complete statement.
```

GET.CURSOR

GET.CURSOR returns the current cursor position and the current cursor shape.

- ☞ The word "cursor" here refers to the blinking line or box that indicates your position on the video screen.

GET.CURSOR (variable)

variable

GET.CURSOR returns four bytes in *variable*. The ASCII values of the bytes indicate the current status of the blinking cursor. The value represented by each byte is:

- 1 the x (column) position (0-79) of the cursor
- 2 the y (row) position (0-24) of the cursor
- 3 the bottom scan line (0-13) for the cursor shape
- 4 the top scan line (0-13) for the cursor shape

For more information, see PUT.CURSOR.

Correct Use of GET.CURSOR

```
CURSOR.INFO = ""
GET.CURSOR( CURSOR.INFO )
COL = SEQ( CURSOR.INFO[1,4] )
ROW = SEQ( CURSOR.INFO[2,4] )
BOT = SEQ( CURSOR.INFO[3,4] )
TOP = SEQ( CURSOR.INFO[4,4] )

TEXT      = " Current cursor attributes"
TEXT<-1> = " _____"
TEXT<-1> = " Column = ":COL
TEXT<-1> = " Row    = ":ROW
TEXT<-1> = " Bottom = ":BOT
TEXT<-1> = " Top    = ":TOP
TEXT<-1> = " Height = ":BOT-TOP
TEXT<-1> = " _____"

MAP = ""
MAP<1> = "A"
MAP<6> = "L"
CALL MSG( TEXT, MAP, "", "" )
STOP
```

The program fetches the current cursor attributes using GET.CURSOR, and then reports the results.

GOSUB

The GOSUB, RETURN, and RETURN TO statements provide the means for subroutine branching; they transfer program control to and from internal subroutines.

```
GOSUB label{:}
```

GOSUB

When R/BASIC encounters a GOSUB statement, it transfers control of the program to the internal subroutine designated by the label. The statements within the subroutine are executed in sequence until a RETURN or RETURN TO is encountered and the control transfers back to the main body of the program.

Use the GOSUB and RETURN statements for branching to and from subroutines rather than the GOTO statements because they make structured programming easier.

label

The *label* may be any valid numeric or alphanumeric label. An alphanumeric *label* must start with an alphabetic character. A colon is optional.

Correct Use of GOSUB

```
PROMPT ""
AVG.TIME = ""
AVG.CNT = 0
PRINT "TYPING TEST"
LOOP
  PRINT "Type the characters displayed:":
  PRINT "Press any key to start.":
  INPUT RESPONSE
  GOSUB GET.CHAR; INPUT RESPONSE
UNTIL RESPONSE EQ "" OR RESPONSE EQ "END"
  GOSUB AVERAGE
  PRINT "You took ":AVG.TIME<AVG.CNT>:
  PRINT " seconds" ; PRINT
REPEAT
GOSUB AVERAGE
STOP

GET.CHAR:
TEST.LINE = ""
FOR I = 1 TO 15
  TEST.LINE := CHAR(RND(43)+48)
NEXT I
PRINT TEST.LINE
BEG.TIME = TIME()
RETURN
```

```

AVERAGE:
IF RESPONSE = "" OR RESPONSE = "END" THEN
  AVG.TOT = 0
  FOR I = 1 TO AVG.CNT
    AVG.TOT = AVG.TOT + AVG.TIME<I>
  NEXT
  PRINT "The average time was ":INT(AVG.TOT / AVG.CNT):
  PRINT " seconds in ":AVG.CNT:" tests."
END ELSE
  AVG.CNT += 1
  IF RESPONSE NE TEST.LINE THEN PENALTY = 5 ELSE PENALTY = 0
  AVG.TIME<AVG.CNT> = TIME() - BEG.TIME + PENALTY
END
RETURN

```

The program branches to subroutine GET.CHAR which produces a line of random characters. When RESPONSE is null or END, the LOOP is finished and the program branches to subroutine AVERAGE.

GOTO

The GOTO statement allows unconditional transfer of program control to any statement within the R/BASIC program.

```

GOTO label{:}
(or)
GO    label{:}

```

label

When R/BASIC encounters a GOTO statement program control is unconditionally transferred to the designated *label*. The statements following that label will be executed. The *label* may be numeric or alphanumeric.

Use the designated label anywhere in the program. Control may be transferred to statements either following or preceding the GOTO statement.

The GOTO statement should be used very carefully to avoid confusion. For structured programming, R/BASIC provides other statements.

:

A colon (:) may be used following the label for clarity and consistency.

- ☞ For more information, see CALL, DECLARE, FOR/NEXT with WHILE or UNTIL, GOSUB, and LOOP.

Correct Use of GOTO

```

PROMPT ""
TOP:
PRINT @(-1)
PRINT @(24,6) : "MENU"
PRINT @(20,8) : "1.  Display current time"
PRINT @(20,9) : "2.  List available files"
PRINT @(20,10) : "3.  Quit"
PRINT @(20,13) : "Enter your choice (1-3):":
INPUT ANS,1
IF ANS = 1 THEN GOTO GET.TIME:
IF ANS = 2 THEN GOTO SHOW.FILES:
IF ANS = 3 THEN PRINT @(-1) ; STOP
* otherwise
PRINT CHAR(7) : @(0,22) :
PRINT "Invalid entry!  Strike any key to reenter:":
INPUT ANS,1
GOTO TOP:
*
GET.TIME:
PRINT @(-1)
EXECUTE "TIME"
GOTO TOP:
*
SHOW.FILES:
PRINT @(-1)
EXECUTE "LISTFILES"
GOTO TOP:
*
END

```

Incorrect Use of GOTO

```
GOTO 5,3
```

Uses two labels where only one is allowed.

HASH

The HASH function takes a record key, modulo, and offset and returns the group number as used in the ROS filing system.

HASH(key, modulo, offset)

key, modulo, offset

HASH determines the group to which a record will HASH. The *key* and *modulo* produce a number between 0 (zero) and modulo minus 1. This number is added to the *offset*, normally 1 (one).

The algorithm used by HASH is exactly the same as that used by Advanced Revelation to read and write records to and from ROS files.

A HASH function is not provided for Linear Hash files.

Correct Use of HASH Function

```
CUST.NO = "101"
GROUP = HASH(CUST.NO, 3, 1)
```

Computes the group into which record 101 would hash in a ROS file with a modulo of 3.

Incorrect Use of HASH Function

```
Z = HASH(file)
```

Only one argument is used where three are required.


```
PRINT HASH (1, 2, 3)
```

Uses space between the function and the parenthesis where no space is allowed.

HEADING

The HEADING statement prints or displays a character string at the top of each page at the printer or the terminal.

HEADING "expression{'options'}::"

Valid Options

Headings are made up of any character string enclosed in double quotes (") and may contain any number of *options*. The options must be further enclosed within single quotes ('). Multiple options may be placed within one set of single (') quotes.

Valid options are:

- | | | |
|------|--------------|---|
| 'B' | Break | Inserts the value of @BREAK into the HEADING. |
| 'D' | Date | Insert the current date as DD MM YYYY. |
| 'F' | Filename | Inserts the current file being read. |
| 'L' | New Line | Causes a line feed. 'LL' will leave one line blank. |
| 'N' | No Page | Suppresses automatic paging. |
| 'P' | Page | Inserts current page number. |
| 'PP' | Page Justify | Inserts current page number with right justification into a four space field. |

"T"	Time	Inserts the current time and date.
" "	Two Single Quotes	Prints a single quote in the heading.

Correct Use of HEADING

```

FLAG = ""
PRINT "INPUT STARTING DATE" ;; INPUT START.DATE
PRINT "INPUT ENDING DATE" ;; INPUT END.DATE
HEADING "WEEKLY REPORT FOR A/P DEPT. 'T' 'P'"
FOOTING " "
OPEN 'INV' TO FILE.INV THEN
  SENT = "SELECT INV"
  SENT := "WITH INV.DATE BETWEEN"
  SENT := QUOTE(START.DATE) : " AND " : QUOTE(END.DATE)
  PERFORM SENT
  LOOP UNTIL FLAG = "STOP"
    READNEXT @ID THEN
      READ REC FROM FILE.INV, @ID THEN
        PRINT @ID:" ":REC<1>:" ":REC<2>:" ":REC<3>
      END ELSE STOP
    END ELSE
      FLAG = "STOP"
  END
REPEAT
END

```

Incorrect Use of HEADING

```

HEADING 'Monthly Report'

```

Uses single quotes where double quotes are required.

ICONV

The ICONV function converts data into the internal system format used by Advanced Revelation.

ICONV(string,conversion)

string

The *string* value represents the data to be converted. This expression is converted to the internal system format by the conversion.

The *string* can specify a dynamic array using field marks (ASCII character 254), value marks (ASCII character 253), subvalue marks (ASCII character 252), or text marks (ASCII character 251) as delimiters.

conversion

The conversion code defined in the *conversion* is enclosed in quotation marks unless it is given as a variable. The result is always a string expression.

After the execution of an ICONV function, the STATUS() function can query the results. Possible status values are:

STATUS()		Meaning
0	=	Successful ICONV
1	=	Bad data string
2	=	Bad conversion definition

For further details on conversion specifications, see:

ICONV Boolean (B)
 ICONV Date (D)
 ICONV Datetime (DT)
 ICONV Masked Decimal (MD)
 ICONV MX, HEX, MO, MB
 ICONV Time (MT)

Correct Use of ICONV Function

```
X = ICONV("100.00", 'MD2')
```

Converts 100.00 to 10000.

```
D = ICONV("9-27-99", "D")
```

Converts September 27, 1999 to its internal date equivalent, 11593.

```
T = ICONV("1:00PM", "MT")
```

Converts 1:00 PM to its internal time equivalent, 46800.

```
PRINT @(-1);* clear screen
```

```
PROMPT ""
```

```
LOOP
```

```
  PRINT 'Enter string to convert: '
```

```
  INPUT STRING
```

```
UNTIL STRING = 'END'
```

```
  PRINT 'Enter the conversion code: '
```

```
  INPUT CODE
```

```
  INTERNAL = ICONV(STRING, CODE)
```

```
  PRINT INTERNAL: ' = ICONV(' : STRING: ', ' : CODE: ' )'
```

```
  EXTERNAL = OCONV(INTERNAL, CODE)
```

```
  PRINT EXTERNAL: ' = OCONV(' : INTERNAL: ', ' : CODE: ' )'
```

```
  PRINT
```

```
REPEAT
```

```
END
```

The program converts the specified string and conversion code to an internal format and assigns the value to INTERNAL. The internal format in INTERNAL is then converted to an external format and assigned to EXTERNAL.

Incorrect Use of ICONV Function

```
NUM = ICONV(3,MD2)
```

The quotes required around the conversion specification are missing.

```
Z = ICONV(2)
```

The conversion specification is missing.

ICONV Boolean (B)

The ICONV boolean function converts characters or words into an internal value of true (1) or false (0).

```
ICONV(string,"B[{tf}  
[true,false}]")
```

string

The *string* contains a character or word that represents a Boolean condition. Examples of characters include “Y” and “N” or “T” and “F”. Examples of words are “Yes” and “No” or “True” and “False”. Any characters or words can be used in *string*. If *string* cannot be converted into a boolean value, STATUS() returns a 1.

B

B indicates a the boolean conversion. If no additional options are provided, the conversion assumes the options “yn”. In other words, “B” is interpreted the same as “Byn”.

tf

When the *tf* option is used, the conversion matches the first character of *string* against the characters represented by *t* and *f*, without regard to case. If the first character of *string* matches *t*, the function returns a Boolean true (1). If the first character matches *f*, a Boolean false (0) is returned. If *string* is null, a null is returned.

true,false

If the option *true,false* is used, the conversion matches *string* against the words represented by *true* and *false*. If *string* matches *true*, a Boolean true is returned. If *string* matches *false*, a Boolean false is returned. For each lowercase alphabetic character in *true,false*, the comparison is done without regard to case. For all other characters (upper case alphabetic or numeric), the comparison is done exactly.

Correct Use of ICONV Boolean (B) Function

```
VALUE = "T"
TEST = ICONV(VALUE, "BTF")
TEST is set to 1 (one).
```

```
RESP = ""
CALL MSG("Do you wish to continue (Y/N)?", "R", RESP, "")
IF ICONV(RESP, "BYN") EQ 0 THEN STOP
```

This program fragment prompts for a response and validates it using the B conversion. If any word beginning with "Y" or "y" is entered, the program continues; if "N" or "n" is entered, the program quits.

```
CONV = "Bsmall,large"
PRINT "Enter the size:":
INPUT SIZE
GENDER = ICONV(SIZE, CONV)
IF STATUS() = 1 THEN
    PRINT "Sorry, that response is incorrect!"
END
```

In this program fragment, the user's response is converted using the pair "small / large". If the user enters "small" (or "SMALL"), SIZE is set to 1. If "large" or "LARGE" is entered, SIZE is set to zero. If an incorrect response is entered, the program prints an error message.

ICONV Date (D)

The ICONV function with Date (D) conversion specifications allows the user to convert the value in a date format into the internal system format for date.

```
ICONV(string, "D{y}{c}{E}{F}{G}{H}{I}{J}")
```

Internal System Date

The internal system date is the number of days since December 31, 1967. This date has been designated internally as day 0 (zero), so all dates after that are represented as the number of days that have elapsed since that specific date. All dates previous to the day 0 (zero) are designated as negative number of days prior to the reference date.

string

The *string* must contain a date format that is compatible with the conversion.

D

D indicates date conversion.

y and c

The *y* and *c* options are allowed but have no effect on the ICONV function. These options have meaning in the OCONV function. They are accepted for input in the ICONV function to allow the same specifications for both input and output. See OCONV Date (D) for more information.

E, F, G, H, I, J

The *E*, *F*, *G*, *H*, *I*, and *J* options interpret the order in which a date (day, month, and year) is read.

Format		Option	
For	01 MAR 1987	use	DE
For	MAR 1987 01	use	DF
For	1987 01 MAR	use	DG
For	MAR 01 1987	use	DH
For	01 1987 MAR	use	DI
For	1987 MAR 01	use	DJ

Different formats of date values are accepted in the input conversion specifications. The day, month, and year may be separated by almost any non-numeric character. If the input data is all numeric, note the order of the month and day. See the *E* option above.

The system clock's current year is assumed if the year value is not input. If only two digits are input for the year value, it is assumed to be the 20th century for values 30-99 (i.e., 1930-1999) and the 21st century for values 0-29 (i.e., 2000-2029). Three digits 100-999 are assumed to be 100-999 AD.

- ☛ If the string is not in the correct format for the conversion, a null will be returned.

For more information, see DATE() and TIMEDATE().

Correct Use of ICONV Date (D) Function

```
PB = ICONV("12/01/92", "D")
PRINT @(20,1) : PB
Prints 9102 at column 20 line 1.

CD = ICONV("01:01:93", "D2:")
Assigns the value 9133 to the variable identifier CD

PRINT ICONV("12.1", "D")
Prints 7275 if it is 1987.
```

Incorrect Use of ICONV Date (D) Function

```
PRINT ICONV(A)
Only one parameter is used where ICONV function requires two.
```

```
PRINT ICONV("12/01/82", "2D")
```

The second parameter should have been D2 and not 2D.

```
ICONV(A, F) = 5000
```

An attempt has been made to assign a value to an internal function.

ICONV Datetime (DT)

The ICONV datetime function converts date and time string into an internal decimal number.

```
ICONV(expression, "DT{dateconv}^{n{c}}{timeconv}")
```

Internal (fractional) Date

The datetime conversion creates a decimal number that represents the number of days (plus fractional days) since December 31, 1967 at midnight. As with the "D" conversion, this date has been chosen to represent day and time zero. Dates previous to December 31, 1967 appear as negative numbers.

Date and time zero (12/31/1967 at midnight) is used as the absolute measure for calculating the fractional value of a day. This means that for negative dates (earlier than the reference date) the fractional portion of the day *increases* as it gets earlier in the day, since this is further away from midnight.

For example, if the date and time is February 19, 1987 at 6:00AM, the internal format is 6990.25 (the .25 indicates one-quarter of the day). The date and time February 19, 1957 at 6:00PM appears as -3966.25. Note that 6:00PM appears as .25 for dates previous to December 31, 1967.

expression

The expression contains a string that evaluates to a date, some delimiter, and a time. The format of the both date and time can be any that are accepted in the ICONV - Date (D) and ICONV - Time (MT) conversions. If the delimiter between the date and the time is not a space, the *c* option must be used.

A space can be used as a delimiter either between the date and the time string, or within the time string, but not in both. For example, the string is valid for the ICONV DT function:

```
01 JAN 1987 10:00
```

because although the delimiter between the date and time is a space, a colon is used in the time string.

☞ For more information on date and time conversions, see the appropriate ICONV function descriptions.

DT

DT indicates the datetime conversion. If no options are specified, the conversion "DT^1 " is assumed.

Options are allowed, but only the *c* option has meaning in the ICONV function. Other options have meaning in the OCONV function. They are accepted for input in the ICONV function to allow the same specifications for both input and output. See OCONV Datetime (DT) for more information.

dateconv

The *dateconv* can be any valid date conversion as documented under the heading ICONV Date (D).

- ☞ Do not include the "D" in the *dateconv* expression. Include only the optional parameters.

^ (caret)

Use the ^ symbol (caret) to delimit the date conversion specification from the time conversion specification.

n

The *n* specifies the number of characters that will separate the date from the time. The default is one. This number is ignored in the ICONV function.

c

If a character other than a space is used to delimit the date from the time in the *expression*, the delimiter character must be specified in *c*. The number of spacing characters is ignored, but the actual character must appear in *c*.

timeconv

The *timeconv* is any valid time conversion as documented under the topic ICONV Time (MT).

- ☞ Do not include the "MT" in the *timeconv* expression. Include only the optional parameters.

For more information, see DATE(), ICONV Date (D), Time (MT), and TIMEDATE().

Correct Use of ICONV Datetime (DT) Specifications

```
TEST = ICONV("02/19/1987 06:00AM", "DT")
```

TEST is set to the value 6990.25.

```
TEST = ICONV("02/19/1987 02:00AM", "DT")
```

TEST is set to the value 6990.0833

```
TEST = ICONV("02/19/1987 12P", "DT")
```

TEST is set to the value 6990.5.

```
TEST = ICONV("02/19/1957 06:00AM", "DT")
```

TEST is set to the value -3966.75.

```
TEST = ICONV("02/19/1957 06:00PM", "DT")
```

TEST is set to the value -3966.25.

```
TEST = ICONV("02/19/1957-06:00PM", "DT^3-")
```

TEST is set to the value -3966.25.

```
TEST = ICONV( DATETIME(), "DT" )
```

TEST is set to the internal (fractional) date and time for the time at which the function is executed.

Incorrect Use of ICONV Datetime (DT) Specifications

```
TEST = ICONV("DT", "02 19 1987 06 00")
```

Spaces are used in both the date and time specifications.

```
TEST = ICONV("DT", "02/19/1987*06:00")
```

The delimiter between the date and time is not a space, but this is not indicated in the "DT" specification.

ICONV MX, HEX, MO, MB

The ICONV function with MX, HEX, MO, or MB conversion specifications converts values from decimal formats to hexadecimal, octal, and binary formats.

```
ICONV(string, "MX")
ICONV(string, "HEX")
ICONV(string, "MO")
ICONV(string, "MB")
```

string

The character range for each character in *string* is 0 - 9, A - F. Using characters outside the respective ranges will result in a conversion error.

MX

The string must be a hexadecimal character set (base 16), and is converted to the equivalent decimal number (base 10).

HEX

The string must be a string of hexadecimal character sets, and is converted to a string of ASCII values.

MO

The string must be an octal (base 8) number, and is converted to a decimal number.

MB

The *string* must be an ASCII representation of a binary (base 2) number, and is converted to a decimal number. If *string* yields an incorrect value, a 0 (zero) will be returned.

- ☛ An understanding of different number bases is required for the effective use of these specifications.

Use the following table as a guide to conversions to different number bases:

Conversion Code	ICONV	OCONV
MX	hexadecimal to decimal	decimal to hexadecimal
MO	octal to decimal	decimal to octal
MB	binary to decimal	decimal to binary

For more information, see OCONV - MX, HEX, MO, MB.

Correct Use of ICONV MX, HEX, MO, MB Specifications

```
PRINT ICONV ("2A", "MX")
```

Output is 42

```
PRINT ICONV ('595A', 'HEX')
```

Output is YZ

```
PRINT ICONV ("1000", "MB")
```

Output is 8

Incorrect Use of ICONV MX, HEX, MO, MB Specifications

```
PRINT ICONV 200 "MX"
```

The parentheses are missing.

```
PRINT ICONV (200, HEX)
```

The quotation marks are missing.

```
Y = ICONV (5)
```

The expression is missing.

ICONV Masked Decimal (MD)

The ICONV function with Masked Decimal (MD) input conversion specifications converts decimal data to internal system representation for storage.

```
ICONV(string,"MDn{f}{,}{$- or < or C or D}{P}{Z}{xc}")
```

string

The *string* must yield a numeric value.

MD

The MD specifies masked decimal conversion.

n

A single digit, 0-9, specifies number of digits to the right of decimal point to be retained. The decimal point will not appear in the data. Excess digits in the decimal column are rounded off.

f

A single digit, 0-9, specifies scaling factor. If *f* is not specified, the default is equal to *n*. The *string* is multiplied by 10 to a factor of *f*. This is equivalent to moving the decimal point one place to the right or adding zeros where there is no decimal point. When OCONV is used, the *string* is divided by 10 to a factor of *f*.

Remaining Options

The remaining options are allowed to retain consistency with OCONV but have no effect on the data conversion. These options have meaning with the OCONV (MD) output conversion function.

The input conversion does not prefix dollar signs or place commas to separate thousands when converting data to internal format.

A negative value results when a minus (-) sign precedes or follows data, when the input data is enclosed in brackets (<...>), or when the DB (debit) or CR (credit) suffix is used.

Invalid input data will be returned as a null.

For more information, see OCONV Masked Decimal (MD).

Correct Use of ICONV Masked Decimal (MD) Specifications

```
PRINT ICONV(103.2,'MD2')
```

Output is 10320.

```
PRINT ICONV(-5, 'MD1')
```

Output is -50.

```
PRINT ICONV(1.234, 'MD2')
```

Output is 123.

```
PRINT ICONV(10, 'MD3')
```

Output is 10000.

```
PRINT ICONV(143.5, 'MD12')
```

Output is 14350.

```
PRINT ICONV(143.5, 'MD14')
```

Output is 1435000.

Incorrect Use of ICONV - Masked Decimal (MD) Specifications

```
X = ICONV(Y)
```

The specifications are missing.

```
ICONV(Y) = 2
```

The function appears on the left side of the assignment operator.

ICONV Masked Scientific (MS)

The ICONV function with Masked Scientific (MS) input conversion specifications converts scientific notation to internal system representation for storage.

```
ICONV(string, "MS{e{mm}}")
```

string

The *string* must yield a valid scientific notation.

MS

Masked Scientific (MS) specifies scientific conversion.

e

The *e* option specifies the number of digits allocated for the power of ten to which the number is raised. The default is 2 (two).

mm

If the *e* option is specified, the *mm* option specifies the mantissa, which is the number of places following the decimal point. The default is six places. The limit for the mantissa is 18. The mantissa may be one or two digits.

Options

Options are allowed but have no effect on the data conversion. These options have meaning with the OCONV (MS) output conversion function.

If the string yields invalid data, a 0 (zero) will be returned.

☛ For more information, see OCONV Masked Scientific (MS).

Correct Use of ICONV Masked Scientific (MS) Specifications

```
PRINT ICONV(1.23, 'MS')
```

Output is 1.23

```
PRINT ICONV(1.2345E+4, 'MS')
```

Output is 12345

```
PRINT ICONV(1.23456789E+2, 'MS')
```

Output is 123.4568

```
PRINT ICONV(1.234E+5, 'MS')
```

Output is 123400

```
PRINT ICONV(1234.1234E-2, 'MS')
```

Output is 12.3412

Incorrect Use of ICONV Masked Scientific (MS) Specifications

```
X = ICONV(Y)
```

Specifications are missing.

```
ICONV(Y) = 2
```

The function appears on the left side of the assignment operator.

ICONV Time (MT)

The ICONV function with Time (MT) conversion specifications converts the data in a time format into the internal system format for time.

```
ICONV(string, "MT{B}{S}{c}")
```

Internal System Time

Internal system time is calculated as the number of seconds past midnight. A 24-hour day has 86,400 seconds.

MT

The MT conversion expression indicates time conversion.

H,S,c

The H, S, and C options are allowed, but they have no effect on the time conversion. These options have meaning with the OCONV function. They are accepted for input in the ICONV function to allow use of the same specifications for both functions.

Different formats of time values are accepted in the input conversion specifications. The hours, minutes, and seconds may be separated by almost any non-numeric character. Use a suffix of AM, PM, A, or P. The 24-hour clock format will be assumed if the hour value is greater than 12. If the minutes and seconds are not input, they will be assumed to be zero.

☛ For more information, see OCONV Time (MT), DATE(), and TIMEDATE().

Correct Use of ICONV Time (MT) Specifications

```
PRINT ICONV("12:01", "MT")
```

Output is 43260.

```
PRINT ICONV("12:01PM", "MTH")
```

Output is 43260.

```
PRINT ICONV("12:01:59PM", "MT")
```

Output is 43319.

```
PRINT ICONV(2, "MT")
```

Output is 7200.

```
PRINT ICONV("2:01AM", "MT")
```

Output is 7260.

```
PRINT ICONV("2:01:59", "MT")
```

Output is 7319.

```
PRINT ICONV("2/01/59", "MT")
```

Output is 7319.

```
A = "08:28PM"
```

```
B = "MTH"
```

```
C = ICONV(A, B)
```

The value 73680 will be stored in C.

Incorrect Use of ICONV Time (MT) Specifications

```
T = ICONV("645")
```

Conversion specification is missing.

```
T = ICONV("12:01", "MH")
```

Illegal specification is used.

IF

The IF statement evaluates the logic of a test expression and makes a conditional transfer based on that evaluation. R/BASIC has three forms of the IF statement: the single line IF statement, the multiple line IF statement, and the conditional expression IF statement. Each of these is described separately on the following pages.

The single line IF statement evaluates the logic of a test expression and makes a conditional transfer to one of two statements depending on the value (true or false) of that test expression.

```
IF test THEN statements {ELSE statements}
(or)
IF test ELSE statements
```

Branching

IF ... THEN ... ELSE statements provide the framework for conditional branching. The normal program sequence is interrupted by testing a condition and then transferring program control depending on the outcome of that evaluation.

test

When R/BASIC encounters an IF statement, the *test* is evaluated as either true (non-zero) or false (zero). If the expression is true, program control will branch to the THEN clause statements. If the *test* is evaluated as false, control will branch to the ELSE clause statements. In this example:

```
IF X = A+B THEN CALL WRAPUP ELSE GOSUB 70
```

The program will call subroutine WRAPUP if X is equal to A+B. The branch will be to subroutine 70 if X is not equal to A+B.

If the *test* is false without an ELSE clause, program control will move on to the next statement that follows the entire IF ... THEN set. If the *test* is false with an ELSE clause, the statements in the ELSE clause will be executed.

More than one statement may be included in the THEN or ELSE clauses; however, the *statement* must be separated by semicolons (;), as in the following example:

```
IF S=500 THEN S=20 ; W=1 ELSE A=S ; GO 10
```


Correct Use of Single Line IF

```
IF X+5 GT Z THEN STOP ELSE GOSUB 100
```

If X + 5 is greater than Z, then the program stops. Otherwise the program branches to subroutine 100.

```
IF A AND B THEN NULL ELSE PRINT A ; GOSUB 300
```

If both A and B are not 0 (zero), then the program executes the next statement. Otherwise the program prints A and branches to subroutine 300.

```
IF A AND B ELSE PRINT A; GOSUB 300
```

This example is the equivalent of the one above.

Incorrect Use of Single Line IF

```
IF T = 90 THEN PRINT "A" GOTO 250
```

A semicolon needed to separate the two statements is missing.

IF (Multiline)

The multiple line IF statement functions the same as the single line IF statement, except that it allows you to write statement sequences on multiple lines.

```
IF test THEN
  statements
END
(or)
IF test THEN
  statements
END ELSE
  statements
END
(or)
IF test THEN statement ELSE
  statements
END
(or)
IF test THEN
  statements
END ELSE statement
```

THEN or ELSE

When THEN or ELSE is the last keyword on a line, R/BASIC assumes that multiple lines follow. The ELSE clause is optional.

One or more statements may appear in the THEN or ELSE clauses; they may be written on the same lines if separated by semicolons.

END

Use END to terminate a block of multiple statements.

Correct Use of Multiple Line IF Statements

```
IF INV.COUNT > 144 THEN
  PRINT INV.COUNT
  GOSUB 250
END ELSE
  PRINT "REORDER"
  GOSUB 400
END
```

If the value of INV.COUNT is more than 144, the program will print the value of INV.COUNT and then transfer to label 250. If the value of INV.COUNT is less or equal to 144, the program will print REORDER and then transfer to subroutine 400 for reordering.

Incorrect Use of Multiple Line IF Statements

```
IF A = 1 THEN
  B = A+1
  C = B+1
ELSE GOTO 1000
```

No END statement is used in this multiline IF statement.

IF (Conditional Expression)

The conditional expression IF functions the same as the single line IF except that it is evaluated as an expression.

```
IF test THEN true.expression {ELSE false.expression}
(or)
IF test ELSE false.expression
```

expression

The conditional expression IF allows the test for a condition where a statement cannot be used. It returns one of two values, depending on the result of the *test* expression.

When the *test* is true, then the value of the *true.expression* is the result of the IF expression. When the *test* is false (0 or null) the value of the *false.expression* is the result of the IF expression. For example:

```
@ANS = IF {CHANGED} THEN "YES" ELSE "NO"
```

The dictionary record that contains the above code is used in a report that shows when a record has been changed. If the CHANGED field is not empty and contains anything but a 0, then "YES" will be printed in the report.

- ✎ The conditional expression IF must be written on a single line. The THEN and ELSE clauses may only contain an expression.

For more information on the use of R/BASIC in dictionaries, see “Interfacing Dictionaries and R/BASIC” in the chapter “Programming with R/BASIC”.

INDEX

The INDEX function examines a string for the occurrence of a specified substring and returns a value that is equal to the starting column position of that specified substring.

INDEX(string, substring, occurrence)

string

A *string* is any legal data, including a null.

substring

The INDEX function searches the string value of the *string* for the substring that is designated by the *substring*.

occurrence

The *occurrence* defines which occurrence of that specified substring is needed.

The INDEX function then returns a numeric value that is the starting column position of that substring within the string expression. In the following example:

```
FIRST = INDEX("ABRACADABRA", "CAD", 1)
```

the variable identifier FIRST is assigned the value of 5 because the first occurrence of the substring “CAD” begins in the column position 5 in the string expression “ABRACADABRA”. In this example:

```
T = INDEX("XXA1YYA1ZZA1", "A1", 3)
```

T would be assigned the numeric value of 11.

If the specified occurrence does not exist, or if the substring is not found, a value of zero is returned.

In the above example, if the last expression had been 4, a 0 (zero) would have resulted because the string A1 does not occur four times. A 0 (zero) would have resulted also if the substring “2” had been used.

Correct Use of INDEX Function

```
T = INDEX("R2D2ME2", 2, 3)
```

T is assigned the value of 7.

```
W = "12L23L34L"
```

```
M = "3L"
```

```
IF INDEX(W,M,1) = 5 THEN
```

```
    CALL WRAPUP(W)
```

```
END
```

The program will call the external subroutine WRAPUP because the substring M does occur starting in the fifth column of the string expression.

Incorrect Use of INDEX Function

```
INDEX("D1D2D3D4", "D2", 1)
```

The function appears alone in the statement.

```
B = INDEX ("123", 2, 1)
```

Uses a space between the function and the parenthesis.

```
B = INDEX("123", 2)
```

The expression denoting the occurrence is missing.

INITDIR

The INITDIR statement uses a operating system pathname to indicate a directory and files to be accessed by the DIRLIST function.

INITDIR expression

INITDIR

INITDIR used in conjunction with DIRLIST() allows a program to access files in remote directories.

expression

The expression used with INITDIR defines the directory path to the files and specifies the type of files that are to be returned.

INITDIR may take the operating system wild cards "*" and "?".

- ☛ For more information on DIR and pathnames, see your operating system manual. See also DIRLIST().

Correct Use of INITDIR Function

```

DRIVE="C:\HISTORY\DOCS\LETTERS\"
INITDIR DRIVE : "*.DOC"
LOOP
  LIST = DIRLIST()
UNTIL LIST = ""
  LOOP
    LINE = LIST[1,@FM]
    LIST[1,COL2()] = ""
    PRINT DRIVE : LINE
  UNTIL LIST = "" REPEAT
REPEAT
END

```

The program gets the list of files in the C:\HISTORY\DOCS\LETTERS\ directory and prints them. It uses INITDIR to indicate the directory and DIRLIST() to return the filenames. Note the use of @FM and COL2() within the brackets.

INITRND

The INITRND statement initializes the random number generator for use with the RND function.

INITRND expression

expression

Use any legal expression. The *expression* is used as a string and it sets a starting value.

The random number generator is initialized by using INITRND preceding the random number procedure. The same series of random numbers is repeated when the procedure is used again.

Correct Use of INITRND

```

PRINT "Enter starting number"
INPUT ANS
INITRND ANS*3
AUTH.NO = 0
FOR I = 1 TO 6
  AUTH.NO := RND(100)
  PRINT @(12,I) : AUTH.NO
NEXT I
END

```

INITRND generates the same sequence of authorization numbers.

```
INITRND TIMEDATE()
```

The present time and date set the random number generator.

Incorrect Use of INITRND

INITRND

The expression is missing.

INMAT()

The INMAT() function returns the number of matrix elements when used with the MATREAD statement.

When used with the OPEN statement, it returns the modulo of an Advanced Revelation ROS file.

INMAT ()

Using INMAT()

INMAT() can be used after a MATREAD statement to determine the number of elements that were loaded into the MATREAD array. If the dynamic array contains more elements than the matrix, INMAT() will return a 0 (zero).

☞ For more information, see MATREAD and OPEN. INMAT() will not return a valid modulo for any file list or ROS.

Correct Use of INMAT() Function

```

OPEN 'CUSTOMER' TO CUST ELSE STOP "No CUSTOMER file."
DIM REC(15)
LOOP
  PRINT "Enter a Record ID : " :
  INPUT KEY
  UNTIL KEY = "END"
    MATREAD REC FROM CUST, KEY THEN
      PRINT "INMAT() = " : INMAT()
      FOR X = 1 TO INMAT()
        PRINT X : " " : CUST(X)
      NEXT X
    END ELSE
      PRINT KEY : " not found."
  END
REPEAT
END

```

The FOR NEXT loop will execute once for each field and print the field number and the field contents.

Incorrect Use of INMAT() Function

```
INMAT ( ) = 5
```

The function appears on the left side of the assignment operator.

```
Q = INMAT (X)
```

Uses an expression where none can be specified.

INP

The INP function inputs a value from a port. An understanding of the hardware is necessary to use this function effectively.

✎ INP is not currently implemented for OS/2

INP (port)

port

The *port* designates the hardware port from which the value will be accepted. The *port* must evaluate to a decimal number between 0 and 65535. The value returned will be a decimal number between 0 and 255.

This function is often used in conjunction with BITAND, BITOR, BITXOR, BITNOT and OUT.

✎ For more information, see the BIT functions.

Correct Use of INP Function

```
PORT = 1305
```

```
PRINT INP (PORT)
```

Data from port number 1305 will be printed.

Incorrect Use of INP Function

```
Q = INP ( )
```

The expression is missing.

```
INP
```

INP is used alone whereas a function cannot be used alone.

INPUT

The INPUT statement displays a screen prompt that signals the operator to enter a value or other data during the execution of a program.

```
INPUT variable{,length}{:}{_}
```

INPUT

When an R/BASIC program encounters an INPUT statement, it displays a prompt character on the terminal screen. This prompt signals the user to input data.

The prompt appears on the terminal screen in response to the INPUT statement. The default prompt is a question mark ("?"). The PROMPT statement may be used to define the prompt. A message may be displayed if the PRINT statement is used, as in this example:

```
PRINT "Enter a value":
INPUT X
```

☞ For more information, see DATA and PROMPT.

variable

The value that is input becomes the value of the *variable* in the INPUT statement. A null value will be assigned if the user responds to the prompt by pressing only the [Enter] key.

length

The *length* specifies the number of characters that will be accepted for input. If the specified number of characters are entered, a carriage return and line feed are automatically executed.

When *length* evaluates to -1, R/BASIC uses a special version of the INPUT statement. This version always immediately returns zero, one, or two characters. The value of the *variable* will be null if no characters from the keyboard are waiting. If any characters are ready, only the next one waiting will be returned.

Unlike the normal INPUT, the backspace key returns as CHAR(8) and the [Enter] key returns as CHAR(13). Pressing a function key or any of the special keys returns two characters. The first character is always CHAR(0) on IBM compatible computers, while the second character gives the key number (scan code) as defined by the operating system. This form of input is generally used inside a loop.

: (colon)

A colon following the INPUT statement suppresses the automatic carriage return and line feed after the data is entered.

_ (underscore)

When *length* is specified, a following underscore rejects any input exceeding *length* and waits for a carriage return.

Correct Use of INPUT

```
INPUT SCORE
```

The value entered in response to the prompt is assigned to SCORE.

```
VAR = " "
EQU F1 TO CHAR(0) : CHAR(59)
EQU CR TO CHAR(13)
LOOP
  INPUT INP,-1
  WHILE INP NE CR DO
    BEGIN CASE
      CASE INP = F1 : GOSUB HELP
      CASE LEN(INP) = 1 : VAR := INP
    END CASE
  REPEAT
  PRINT VAR
  STOP
```

```
HELP:
* (help logic here)
RETURN
```

The program will continue looping and concatenating characters until [F1] or [Enter] is pressed.

Incorrect Use of INPUT

```
INPUT A,C,D
```

Three variables are used where only one is allowed.

INSERT

The INSERT function inserts a field, a value, or a subvalue along with a corresponding character mark into a character string dynamic array.

INSERT (string, field, value, subvalue, new)

string

The *string* designates the dynamic array that is to be searched.

field, value, subvalue

The second, third, and fourth expressions are the delimiters. Their respective numeric values determine whether the new data is inserted as a field, a value, or a subvalue. For instance, this example statement:

```
F = INSERT (A, 2, 0, 0, NEW)
```

inserts NEW as a field. When both the value and subvalue are 0 (zero), the new data is inserted before the second field (specified by the field) of dynamic array A. F is assigned the new array.

This example:

```
V = INSERT (A, 2, 3, 0, NEW)
```

inserts NEW as a value. When only the *subvalue* is 0 (zero), the new data is inserted before the third value of the second field (specified by the *value*) of dynamic array A. V is assigned the new entire array.

This example:

```
S = INSERT (A, 2, 3, 1, NEW)
```

inserts NEW as a subvalue. When all three delimiter expressions have a non-zero value, the new data is inserted before the first subvalue of the second value of the third field (specified by the *subvalue*) of dynamic array A. S is assigned the new array.

If the second, third, or fourth expression has a -1 (minus one) value, the new data is inserted after the specified field, value, or subvalue delimiter. For example:

```
F = INSERT (A, -1, 0, 0, NEW)
```

appends NEW as a field to the end of the array.

Field is the highest level delimiter while subvalue is the lowest level delimiter. If a higher level delimiter has a 0 (zero) value while a lower level delimiter has a non-zero value, the zero delimiter is assumed to be 1 (one). As in this example:

```
INSERT (A, 0, 0, 2, B)
```

is assumed to be

```
INSERT (A, 1, 1, 2, B)
```

✎ The *string* is not modified by INSERT.

For more information, see DELETE, EXTRACT, and REPLACE. See also “Dynamic Arrays” in the chapter “Programming with R/BASIC”.

new

The last expression specifies the new data that is to be inserted. The corresponding character mark is also inserted.

Correct Use of INSERT Function

```
A = ""
PRINT "Create a dynamic array!"
LOOP
  PRINT "Enter field position (END to quit): "; INPUT FIELD
  UNTIL FIELD = "END" DO
    PRINT "Enter value position      : "; INPUT VALUE
    PRINT "Enter subvalue position : "; INPUT SUB
    PRINT "Now enter new data       : "; INPUT NEW
    PRINT "Before : ":A
    A = INSERT(A, FIELD, VALUE, SUB, NEW)
    PRINT "After  : ":A
  REPEAT
END
```

The program prompts the user for the field, value, and subvalue positions, as well as the data to be inserted before them. The array is then printed before and after the insertion to illustrate the effect.

INT

The INT function returns an integer numeric value when calculating an expression.

INT(expression)

expression

The *expression* may yield any numeric value. INT truncates any fractional portion of an expression. Arithmetic operations that are specified are calculated using 15 digit accuracy, but INT truncates, not rounds, the decimal fraction. In the following example:

```
MILES = INT (59.99)
```

the value of 59 is assigned to MILES.

If X is any number, then INT(X) is the largest integer less than or equal to X.

Expression	Result
INT(6.23)	is equal to 6.
INT(9.99)	is equal to 9.
INT(100.001)	is equal to 100.
INT(-6.3)	is equal to -7.

To convert X from dollars to cents, the following expression can be used:

```
CENTS = INT(X*100)
```

☞ All truncations are toward the minus infinity.

Correct Use of INT Function

```
I = INT(9/5)
```

Assigns the value of 1 to the variable I.

```
SUM = INT(9.9+.9)
```

Assigns the value of 10 to the variable SUM.

```
AVG = 59/12
```

```
RATE = INT(AVG)
```

Assigns the value of 4 to the variable RATE.

```
PRINT "    X    INT(X)"
```

```
PRINT " ===== "
```

```
FOR X = 2 TO -2 STEP -.25
```

```
    PRINT X, INT(X)
```

```
NEXT X
```

The value for X decreases by .25 until -2 is reached. The integer of each step is printed.

Incorrect Use of INT Function

```
Y3 = "1st"
```

```
Y4 = INT(Y3)
```

Expression in the INT function is not numeric.

```
INT(1) = "DONE"
```

The function appears on the left side of the assignment operator.

```
Q = INT(X, Y)
```

Uses more than one expression.

LEN

The LEN function determines the number of characters in a string and returns that numeric value.

LEN(expression)

expression

LEN calculates the number of characters in a string data value specified by the *expression*, and assigns a numeric value. For example:

```
A = "9876"  
B = "XYZ"  
C = LEN(A : B)
```

assigns the value of 7 to the variable identifier C.

Trailing blanks are counted in the conversion, as in:

```
X = "500 "  
Y = "CREDIT LINE"  
Z = LEN(X:Y)
```

The numeric value of 15 is assigned to the variable identifier Z, which includes the space following 500.

Correct Use of LEN Function

```
J = "CD.#290B"  
PRINT LEN(J)  
Prints a value of 8.
```

```
STRING = "342+431+000"  
NN = LEN(STRING)  
The variable identifier NN is assigned the value of 11.
```

```
PRINT LEN("")  
Prints a 0 (zero).
```

Incorrect Use of LEN Function

```
D = LEN("S", 20)  
Uses more than one expression.
```

```
LEN(X) = Y : Z  
The function appears on the left side of the assignment operator.
```

```
T = LEN (A)
```

Uses space between the function and the parenthesis.

LINEMARK

The LINEMARK statement allows the Debugger to be called at specific lines in R/BASIC programs compiled with the L option.

LINEMARK

LINEMARK

Use LINEMARK only in connection with a program compiled with the L option. The L option suppresses the generation of linemarks in R/BASIC object code, reducing the size.

LINEMARK forces the generation of a LINEMARK opcode, allowing the Debugger to be called at that line.

If LINEMARK is not used, the object code is read as a single line. If the Debugger is called, it can only break into line 1.

Compiling a program without the L option will generate a linemark opcode for each line of source code. The Debugger is called at the first linemark opcode following DEBUG or [Ctrl+Break].

Correct Use of LINEMARK

```
FOR I = 1 TO 100  
  PRINT I  
  LINEMARK  
NEXT I
```

When LINEMARK is used within a loop, the Debugger may be entered at every iteration of the loop.

LN

The LN (natural logarithm) function returns the natural (base e) logarithm of an expression.

LN(expression)

LN

LN calculates the natural logarithm to the (base e) of the value of the *expression*. Base e is approximately 2.71828. If the *expression* evaluates to 0 (zero) or less than 0 (zero), an error message will be printed.

For more information, see EXP.

Correct Use of LN Function

```
PRINT LN(X)
```

Prints the natural logarithm of X.

```
DELTA = X-EXP (LN (X) )
```

Sets variable identifier DELTA to the internal calculation error of the natural log and its inverse.

Incorrect Use of LN Function

```
LN (B) = Y
```

The function appears on the left side of the assignment operator.

```
Q = LN+5
```

The expression is missing.

LOCATE

The LOCATE statement searches a string to locate the position (index) of a substring that contains specified data.

```
LOCATE substring IN string {USING delim} SETTING start  ⇐  
THEN|ELSE statements
```

substring, string

The *substring* specifies the value whose position is to be located in the *string*. The expression may be a variable identifier, a literal string, an array element, a function, or a null. The *string* following IN designates the *string* that is to be searched.

delim

The *delim* specifies the character that is to be used in the search for the *substring*. It may be any ASCII character. If dynamic arrays are being searched, the *delim* should be a field mark (ASCII character 254), value mark (ASCII character 253), or a subvalue mark (ASCII character 252). If a USING clause is not specified, a value mark is assumed.

Do not include a delim character in a string expression. When LOCATE searches a multivalued string, the string following LOCATE should not contain any value

marks (ASCII character 253); likewise, when a subvalued string is designated, the substring should not contain any subvalue marks (ASCII character 252).

start

When the *substring* has been found in the *string*, the number that corresponds to its position in the *string* is assigned to the variable specified in the SETTING clause. The number assigned will relate to the last parameter that was specified in *delim*. That is, if a field mark is specified, the field number is assigned; if a value mark is specified, the value number is assigned; if a subvalue mark is specified, the subvalue number is assigned.

If the *substring* following LOCATE cannot be found, the SETTING variable identifier will be assigned a value of one greater than the number of positions in the string, and the ELSE statement is executed.

THEN

The statements following THEN are executed if the first string is found.

ELSE

If the first string is not found, the statements following ELSE are executed.

☞ For more information about the use of THEN and ELSE, see IF.

For more information, see [] (bracket), EXTRACT, FIELD, FIELDSTORE, INDEX, INSERT, and REPLACE. For information on string handling, see "Dynamic String Arrays" in the chapter "Programming with R/BASIC".

Correct Use of LOCATE

```
CUST.NR = ""; TOT = ""
OPEN "INV" TO INV.FILE ELSE STOP "NO FILE"
SELECT INV.FILE
LOOP
  READNEXT KEY THEN
    READ INV.REC FROM INV.FILE, KEY THEN
      NR = INV.REC<2>
      AMT = INV.REC<3>
      LOCATE NR IN CUST.NR USING @FM SETTING POS THEN
        TOT<POS> = TOT<POS> + AMT
      END ELSE
        CUST.NR<POS> = NR
        TOT<POS> = TOT<POS> + AMT
      END
    END ELSE
      PRINT "NO RECORD AT KEY ":KEY
    END
  END ELSE
    KEY = "DONE"
  END
UNTIL KEY = "DONE" REPEAT
```



```

CNT = COUNT(CUST.NR,@FM) + (CUST.NR NE "")
FOR I = 1 TO CNT
    PRINT CUST.NR<1>, TOT<1>
NEXT I
END

```

The program accumulates totals using dynamic arrays.

LOCATE BY

The LOCATE statement may be used with an optional BY clause. The BY clause locates the sorted index position of a field, value, or subvalue.

```

LOCATE substring IN string BY seq USING expr ⇨
    SETTING var THEN|ELSE statements

```

seq, substring

LOCATE uses *seq* to determine placement of the *substring*. The BY clause must follow the IN clause in this general format. *Seq* may have any of the following values:

Value	Justification
AL	ascending, left justified
AR	ascending, right justified (numeric)
DL	descending, left justified
DR	descending, right justified (numeric)

BY Clause

The BY clause assumes that *string* is already sorted by the *seq* parameter. LOCATE stops scanning the *string* if the search data is not found in the expected sorted location. The variable given in the SETTING clause is set to the index value where the data should be inserted.

LOCATE never changes the *string*. It only scans the *string* and leaves the updating of the string to you.

For more information, see LOCATE.

Correct Use of LOCATE BY

```

C.ARY=""; C.KEYS.POS=""; C.KEYS=""
OPEN "CST" TO CUST.FILE ELSE STOP "NO FILE"
SELECT CUST.FILE
LOOP
    READNEXT KEY THEN
        READ CUST.REC FROM CUST.FILE, KEY THEN
            CITY = CUST.REC<5>
            LOCATE CITY IN C.ARY BY "AL" USING @FM SETTING C.POS ELSE
                C.ARY = INSERT(C.ARY,C.POS,0,0,CITY)

```

```

        END
        GOSUB LOCATE.CUST.NBR: ; * occurs on pos & neg find
    END
END ELSE
    KEY = "DONE"
END
UNTIL KEY = "DONE" REPEAT
STOP
LOCATE.CUST.NBR:
LOCATE KEY IN C.KEYS.POS BY "AR" USING @VM SETTING K.POS ELSE
    C.KEYS = INSERT(C.KEYS,C.POS,K.POS,0,KEY)
END
RETURN
END

```

The program creates a sorted dynamic array placing a city in each field. Within each field is a sorted array of customers for that city.

LOCK

The LOCK statement sets locks that coordinate access to files and records on a network system.

LOCK [filevar [CURSOR cursorvar]] { , key { , locktype } } THEN ELSE statements

LOCK

LOCK alerts the network system that the program wishes to establish a lock on a file or record. If the program attempts to set a lock that has been previously set by another user, the current program will execute the ELSE statements.

Used properly, LOCK allows only one user in a network to access any specific file or record. When a file or record is locked by a user in a network, no other user in the network can lock that record until it has been unlocked using UNLOCK.

LOCK has no effect unless used on a local area network *and with a file type that supports locking*. Therefore, LOCK and UNLOCK can be built into applications whether or not they will be used on network systems. Applications not running on a network will simply ignore the LOCK and UNLOCK commands.

Termination of the program does not unlock any locks that have been set. Use UNLOCK to release each LOCK, or UNLOCK ALL to release all locks currently set. Logging off the system will release all locks. In addition, all locks are cleared when a user resets the system using the TCL RESET command or the END command at the debugger prompt.

- ☛ Locking a file or record is not a guarantee that it will not be changed by another user. All programs must contain LOCK commands to ensure that files or records are not simultaneously updated.

filevar

The *filevar* identifies the file created by a previous OPEN statement.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement. For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

key

If a record is to be locked, the *key* specifies which record in a file to lock. To lock an entire file, use the keyword ALL (no quotes), or pass a null as the key expression.

locktype

The *locktype* specifies the type of locking that is to take place. *Locktype* can be passed as a mnemonic code (not a literal). Valid *locktype* mnemonic codes are:

Code	Meaning
X	Exclusive lock
S	Shared lock
XC	Exclusive coordinated lock (record lock only)
SC	Shared coordinated lock (record lock only)

Locktype can also be passed as a two-digit numeric code (not a literal). Valid numeric codes for *locktype* are:

First Digit Code	Meaning
0	Exclusive lock
1	Shared lock
2	Exclusive coordinated lock (record lock only)
3	Shared coordinated lock (record lock only)

Second Digit Code	Meaning
0	ALL (UNLOCK ALL only)
1	File lock
2	Record lock

The default value for *locktype* is an exclusive lock. Coordinated locking is controlled through the "Coordinated Locking" prompt in the Global Environment Window. For more information about locking and lock types, see *Networking*.

THEN

The *statement(s)* following THEN are executed if a file or record is locked successfully or if a lock is attempted against a file type that does not support locking.

☛ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the file or record cannot be locked. The STATUS() function indicates the severity of the error (see below), and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

STATUS()

The definition of the return value of STATUS() on the ELSE branch of the LOCK statement is as follows:

Value	Meaning
0	Unsuccessful lock on a network; not the station's own lock.
1	Unsuccessful lock; current station's previous lock.

For more information, see UNLOCK.

Correct Use of LOCK

```
LOCKED = 0
LOOP
  LOCK FILE_VAR, @ID THEN
    LOCKED = 1
    FOR CTR = 1 TO 1000
    NEXT
  END
```

```
UNTIL LOCKED REPEAT
```

This program fragment loops until the target record is locked. Note the use of an empty loop as a delay between lock attempts. Because no other lock option is specified, the lock will be an exclusive record lock.

```
DECLARE SUBROUTINE MSG, FMSG
EQU TRUE$ TO 1
EQU FALSE$ TO 0
EQU ESC$ TO \1B\
OPEN "CUSTOMER" TO CUST_FILE ELSE
  MSG("No CUSTOMER file available!", "", "", "")
  STOP
END
LOOP
  KEY = ""
  MSG("Enter a record key (END to quit)", "R", KEY, "")
  UNTIL KEY = "END"
  LOOP
    RETRY = FALSE$
    LOCK CUST_FILE, KEY, S THEN
      READ CUST_REC FROM CUST_FILE, KEY ELSE
        FMSG()
```

```

        END
        UNLOCK CUST_FILE, KEY ELSE
            FMSG()
        END
    END ELSE
        IF STATUS() THEN
            MSG("Locked by you","", "", "")
        END ELSE
            TEXT = "Locked by another station"
            TEXT<1> = "[Esc] to quit, any other key to retry:"
            ANS = ""
            MSG(TEXT, "RE", ANS, "")
            IF ANS NE ESC$ THEN RETRY = TRUE$
        END
    END
    WHILE RETRY REPEAT
    REPEAT
    STOP

```

This program attempts to lock a record in the CST file. If the record cannot be locked, the program displays a message. The user can quit the program, or can attempt the lock again. After the record has been read, it is unlocked. Because the record will not be updated, a shared lock is set.

```

* an exclusive file lock
DECLARE SUBROUTINE MSG
OPEN FILE TO FILE_VAR ELSE STOP
LOCK FILE VAR, ALL, 01 THEN
    MSG("File locked OK","", "", "")
END ELSE
    TEXT = "Lock status = ":STATUS()
    MSG(TEXT, "", "", "")
END

```

Incorrect Use of LOCK

```
LOCK ELSE
```

File and key are missing

```
LOCK FILE_VAR, ALL, 21 THEN ...
```

Exclusive coordinated locks cannot be set using a file lock.

LOOP

The LOOP statement provides a vehicle for structured program looping. It marks the beginning of a group of statements to be executed repeatedly within the conditions set by a test expression.

```

LOOP
  {statements}
  { [WHILE test] {DO statements}
  [UNTIL      ]
    {statements}
  REPEAT

```

statements

Any *statements* that follow LOOP are initially executed, and then any WHILE or UNTIL *test* is evaluated to determine whether its value is true (non-zero) or false (zero).

WHILE

If the WHILE format is used and *test* evaluates to non-zero, any *statements* following DO will be executed and the program will continue to repeat until the *test* is evaluated to false. When the *test* does evaluate to false, program control passes immediately out of the loop and the next statement after REPEAT is executed.

UNTIL

If the UNTIL format is used and the *test* evaluates to non-zero, the program control passes immediately out of the loop to the next statement after REPEAT. But as long as the *test* evaluates to false, that is, until a certain specification has been met, the program will continue to loop.

☞ LOOP WHILE true.
 LOOP UNTIL false.

In the following example

```
LOOP;T=T+1;WHILE T < 50 DO;PRINT T;REPEAT
```

the value of T will be printed and the program will continue to loop until the value of T is more than 49.

The DO following the UNTIL and WHILE syntaxes is optional. You may use any number of UNTIL and WHILE blocks.

Correct Use of LOOP

```

* PRE TEST LOOP
X = ""
LOOP UNTIL LEN(X) > 0
  INPUT X,-1
REPEAT
PRINT "KEY CODE IS ":SEQ(X[1,1])
END

```

The program will loop until the first key is pressed. The ASCII sequence of the first character will be printed.

```
* POST TEST LOOP
X = ""
LOOP
  INPUT X,-1
  WHILE LEN(X) = 0
  REPEAT
  PRINT "KEY CODE IS ":SEQ(X[1,1])
  END
```

The second program will also loop until the first key is pressed. The sequence of the first character will be printed.

```
* INTER TEST FOR LOOP
TOTAL = 0
LOOP
  PRINT "ENTER AMOUNT " :
  INPUT AMOUNT
  UNTIL AMOUNT = "END"
  TOTAL += AMOUNT
  REPEAT
  PRINT "TOTAL AMOUNT ":TOTAL:
  END
```

The third program will accumulate TOTAL until END is typed. The contents of TOTAL will be printed.

MAT

The MAT statement assigns a value to each element in a matrix. All elements may be assigned the same value or the elements may be set equal to the corresponding values in a second matrix.

MAT matrix = $\begin{bmatrix} \text{expression} \\ \text{MAT matrix2} \end{bmatrix}$

matrix

The *matrix* specifies the name of the matrix whose elements are being assigned. The *matrix* must have been previously named and dimensioned in a DIMENSION or COMMON statement.

expression

The value of the *expression* is assigned to each of the elements in the matrix. Any

legal expression may be used. In the following example, all the elements of array X are assigned the current value of A*B+C:

```
MAT X = A*B+C.
```

matrix2

The values of the elements of *matrix2* are assigned to the elements of *matrix*. Both matrices must have been named and dimensioned, and they must contain exactly the same number of elements. The dimensions may differ, but the total number of elements must be the same. For example, X(20), Y(4,5), and Z(2,10) are dimensioned differently, but have the same number of elements.

The elements are assigned in corresponding order: the value of the *i*th element of *matrix2* is assigned to the *i*th element of *matrix*. The elements of *matrix2* remain unchanged.

Matrix Passing

MAT may be used to pass an array to a subroutine or a function. The MAT statement is part of the argument list and is separated from any other arguments by a comma. Both the calling program and the subroutine or function must use the MAT statement as part of the argument list. The matrix must be dimensioned before it is passed.

- ☛ For more information, see CALL, COMMON, DECLARE, DIMENSION, FUNCTION, and SUBROUTINE. See also “External Functions and Subroutines” in the chapter “Programming with R/BASIC”.

Correct Use of MAT Statement

```
DIM MATRIX1(3,4), MATRIX2(12)
FOR CTR = 1 TO 12
  Y(CTR) = CTR
NEXT CTR
MAT MATRIX1 = MAT MATRIX2
MAT MATRIX2 = ""
```

The element values in matrix MATRIX2 will be assigned to matrix MATRIX1 such that MATRIX1(0) would have 0, MATRIX1(1,1) would have 1, MATRIX1(1,2) would be 2, and so forth. Then all elements of MATRIX2 are set to null.

```
CALL CREDIT(MAT CUST.REC, YR, MO)
```

The subroutine CREDIT is called. The matrix CUST.REC and the variables YR and MO are passed as arguments.

Incorrect Use of MAT Statement

```
MAT = P*Q N
```

No matrix is named.


```
DIM V(21)
MAT V = MAT W
W is not a matrix.
```

```
MAT C = MAT A * MAT B
MAT is used in an expression.
```

MATCHES

The MATCHES operator compares a string value to a predefined pattern and evaluates to 1 (one), if true, and 0 (zero), if false.

<code>string</code>	<code>[MATCH MATCHES]</code>	<code>pattern</code>
---------------------	----------------------------------	----------------------

Using MATCH

MATCH is a comparison operator. It compares the *string* value of the first expression to the defined *pattern* specified by the second expression. If the patterns match, the evaluation is 1 (true); if they do not match, the evaluation will be 0 (false). The values that are compared are the number of characters and the data types specified.

For example, if the string value of X in the expression:

```
X MATCH "5N"
```

consists of five numbers, it will evaluate to 1 (true).

The chart following lists the possible pattern specifications.

pattern

The *pattern* must be enclosed in quotation marks, or it must be a variable identifier. *Pattern* specifications are evaluated from left to right. The maximum length permitted for the string is 254 characters; the *pattern* specification expression may not exceed 254 characters.

Pattern Specifications

Pattern	Meaning
nA	An integer followed by A tests for that number of alphabetic characters
nN	An integer followed by N tests for that number of numeric characters

Pattern	Meaning
nX	An integer followed by X tests for that number of any characters
nZ	An integer followed by Z tests for up to that number of characters
"string"	A literal string enclosed in quotes tests for that exact text

☞ If *n* is zero, the match is interpreted to mean any number of that pattern. For example, the pattern 0A indicates any number of alphabetic characters.

Correct Use of MATCHES Operator

```
PHONE = "(206)226-9362"
PHONE MATCH "' (' 3N' ) ' 3N' - ' 4N"
Expression will evaluate to 0 (false) because the phone number has no spaces around the dash or the parentheses.

CN = "B1234WA"
CN MATCHES "1A4N2A"
Expression will evaluate to 1 (true).
```

MATPARSE

The MATPARSE statement assigns the value of each successive field in a dynamic array to successive elements in a matrix.

```
MATPARSE expression INTO matrix {USING delimiter}
```

expression

Any legal expression that yields a string containing delimiters.

matrix

The designated *matrix* must have been previously named and dimensioned by either a DIMENSION or a COMMON statement.

The data contents of each field in the dynamic array will become the data contents of an element in the *matrix*.

The first field will be assigned to the first element, the second field will be assigned to the second element and so on. If the dynamic array has more fields than the *matrix* has elements, the remaining fields will be assigned to the last element of the array as a dynamic array.

delimiter

The *delimiter* specifies the character that is to be used in the assignment of string elements to *matrix* elements. If dynamic arrays are being searched, the *delimiter* should be a field mark (ASCII character 254), a value mark (ASCII character 253), or a subvalue mark (ASCII character 252). If a USING clause is not specified, a field mark is assumed.

Correct Use of MATPARSE Statement

```
DIM INV.KEYS(20)
PROMPT ""
OPEN "CST" TO FILE.CUST ELSE STOP "NO CUST"
OPEN "INV" TO FILE.INV ELSE STOP "NO INV"
PRINT "Enter CUST code:":
INPUT KEY
READ CUST.REC FROM FILE.CUST, KEY ELSE
  STOP "No CUST record at ":"KEY
END
INV = CUST.REC<5>
CONVERT @VM TO @FM IN INV
MATPARSE INV INTO INV.KEYS
PRINT "DATE" : SPACE(7) : "INVOICE AMOUNT"
FOR I = 1 TO 20 UNTIL INV.KEYS(I) = ""
  READ LINE FROM FILE.INV, INV.KEYS(I) THEN
    PRINT OCONV(LINE<2>, "D2-") : SPACE(3) : ⇨
    OCONV(LINE<6>, "MD2,") "R#7"
  END
NEXT I
END
```

The invoice keys are kept in field 5 in the customer record. The value marks are converted to field marks. The string is then parsed into the INV.KEYS matrix. The invoice dates and amounts are then printed from the LINE record that contains the invoice.

Incorrect Use of MATPARSE Statement

```
STRING = 1 : @VM : 2 : @VM : 3
MATPARSE STRING INTO MATRIX1
```

MATPARSE parses on field marks, not value marks, unless a USING clause is specified.

MATREAD

The MATREAD statement reads a record from an Advanced Revelation file and assigns the value of each successive field to consecutive elements of a matrix.

MATREAD matrix FROM filevar CURSOR cursorvar ,key THEN ELSE statements
--

matrix

The designated *matrix* must have been previously named and dimensioned by either a DIMENSION or COMMON statement.

The data content of the fields that are read from the specified record will become the data in the designated *matrix*. Data that is stored in the fields of a record are assigned as elements of the matrix in an ordered manner: the data of the first field in the record is assigned to the first element of the matrix, the data of the second field is assigned to the second element and so on.

☞ MATREAD does not assign a value to the 0 (zero) element of the matrix.

If the record that is read contains fewer fields than the matrix has elements, the extra elements will be assigned a null value. If the record contains more fields than the matrix has elements, the remaining fields will be assigned to the last element of the matrix as a single string, retaining any outstanding field marks.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement, and must be preceded with the word CURSOR.

If the file being accessed has had control features added, cursor access will automatically invoke data (domain) conversion during a MATREAD operation.

☞ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

filevar

The *filevar* must refer to a file variable name that has been previously named in an OPEN statement.

If the specified file cannot be accessed, the program aborts with a runtime error message, and the elements of the *matrix* are left unchanged.

key

The record referenced by *key* will be read from the file identified by *filevar* or the file accessed using the cursor in *cursorvar*.

THEN

The *statement(s)* following THEN are executed if the record can be read successfully. The value of the INMAT() function will be set to the number of elements in the matrix that received values in the execution of the MATREAD statement.

☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the record cannot be read. The elements of the matrix are left unchanged. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

- ✎ For more information on dimensioned arrays, see “Matrices” in the chapter “Programming with R/BASIC”. See also MAT and MATWRITE.

Correct Use of MATREAD Statement

```

DECLARE SUBROUTINE MSG, FMSG
OPEN "CUSTOMER" TO CUST FILE ELSE
  MSG("201", "", "", "CUSTOMER") ; * file not attached error
  STOP
END
KEY = "C932"
DIM CUST(30)
MATREAD CUST FROM CUST_FILE, KEY ELSE
  FMSG()
END
FOR CTR = 1 TO 30
  MSG("Next element = %1%", "", "", CUST(CTR))
NEXT CTR
STOP

```

The CUSTOMER record is read into the CUST matrix. Each field is in one element.

MATUNPARSE

The MATUNPARSE function assigns the value of each successive element in a matrix to successive elements in a dynamic array.

MATUNPARSE (matrix)

matrix

The *matrix* is any matrix name. The *matrix* name must be dimensioned by either a DIMENSION or a COMMON statement before using MATUNPARSE. The data contents of each element in the *matrix* will become the data contents of a field in the dynamic array.

The first element will be assigned to the first field, the second element will be assigned to the second field and so on. If there is a dynamic array in a matrix element, it is assigned as a unit when that element is assigned to the variable.

MATUNPARSE is useful when the contents of an undersized matrix are to be moved to a correctly sized matrix.

Correct Use of MATUNPARSE Function

```

NEW = 0
DIM REC(2)
INPUT FILE.NAME
INPUT REC.NAME
OPEN FILE.NAME TO FILE ELSE STOP 'NO FILE'
MATREAD REC FROM FILE,REC.NAME ELSE STOP 'NO REC'
PRINT "REC(2) = ":REC(2)
CT = COUNT(REC(2),@FM)
IF CT > 0 THEN
    TEMP = MATUNPARSE(REC)
    PRINT "TEMP = ":TEMP
    DIM NEWREC(2+CT)
    MATPARSE TEMP INTO NEWREC
    PRINT "NEWREC(3) = ":NEWREC(3)
    NEW = 1
END

```

The last element in the matrix REC will contain field marks if the dynamic array has more elements. If the record read has more elements, then the matrix is unparsed and a new matrix is dimensioned to fit the size of the record. The dynamic array TEMP is then parsed into NEWREC. Setting NEW may be used to test if a resize occurred.

Incorrect Use of MATUNPARSE Function

```
DYNAM = MATUNPARSE(CUST(2))
```

A subscript is used with the matrix.

MATWRITE

The MATWRITE statement writes the values of the elements of a designated matrix into a record.

MATWRITE <i>matrix</i> ON filevar CURSOR cursorvar ,key THEN ELSE statements

matrix

The *matrix* must have been previously named and dimensioned by either a DIMENSION or a COMMON statement.

All the elements of the selected *matrix* are written as a single record in the designated file in an ordered manner: the data of the first element in the *matrix* is assigned to the first field, the data of the second element is assigned to the second field, and so on.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement, and must be preceded with the word CURSOR.

If the file being accessed has had control features added, cursor access will automatically use domain validation and conversion during a MATWRITE.

- ☞ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

filevar

The *filevar* must refer to a file variable name that has been previously named in an OPEN statement. If the program cannot access the specified file, it aborts with a runtime error message, and the value of the variable is left unchanged.

key

The record referenced by *key* will be written to the file identified by *filevar* or the file accessed using the cursor in *cursorvar*.

- ☞ For more information, see MATREAD and MAT. See also “Matrices” in the chapter “Programming with R/BASIC”.

THEN

The *statement(s)* following THEN are executed if matrix is written successfully.

- ☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the matrix cannot be written. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

- ☞ For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in a MATWRITE statement are optional. However, it is strongly recommended that all write logic be constructed to accommodate possible unsuccessful writes. If no THEN or ELSE clause is specified, any errors during the MATWRITE operation are passed to the system error handling routine.

For more information, see OPEN, MATREAD, READ, READV, WRITE, and WRITEV.

Correct Use of MATWRITE

```

DIM CUST(30)
FOR CTR = 1 TO 30
    CUST(CTR) = CTR
NEXT CTR
MATWRITE CUST ON FILE_CUST_FILE, CUST NO ELSE
    CALL FSMSG("Unable to write %1%!", CUST.NO)
END

```

The matrix CUST is written to file. Each array element becomes a field in the record.

Incorrect Use of MATWRITE

```

MATWRITE MP NEWFILE, "TEST" ELSE
    CALL FSMSG()
END

```

The expression is not preceded by ON or TO.

```

MATWRITE A ON B "SALES"

```

No comma follows the file variable B.

MOD

The MOD function returns the modulo remainder of two expressions.

```

MOD (expression, expression)
    (or)
REM (expression, expression)
REM

```

Using MOD

The modulo (remainder) is calculated by the following formula:

$$\text{MOD}(X, Y) = X - (\text{INT}(X/Y) * Y)$$

expressions

The first *expression*, X, is divided by Y and the INT function truncates the quotient. The resulting integer is multiplied by Y, the second *expression*, and that result is subtracted from the first *expression* X.

Note the following example results in the value 4 assigned to the variable identifier COST:

```

COST = MOD(16, 6)

```

The *expressions* must be of numeric value, and the second *expression* cannot be 0 (zero).

MOD will calculate a remainder after division by a divisor. This is useful when working with complete and incomplete sets.

Correct Use of MOD Function

```
A = 99; B = 10
VAL = MOD(A, B)
```

The variable identifier VAL is assigned the value of 9.

```
J = 49; K = 5
NUM = MOD(J, K)
```

The variable identifier NUM is assigned the value of 4.

```
DATE = DATE()
DAY = MOD(DATE, 7)
BEGIN CASE
  CASE DAY = 0 ; DAY = "SUNDAY"
  CASE DAY = 1 ; DAY = "MONDAY"
  CASE DAY = 2 ; DAY = "TUESDAY"
  CASE DAY = 3 ; DAY = "WEDNESDAY"
  CASE DAY = 4 ; DAY = "THURSDAY"
  CASE DAY = 5 ; DAY = "FRIDAY"
  CASE DAY = 6 ; DAY = "SATURDAY"
END CASE
@ANS = DAY
PRINT @ANS
END
```

The system date is returned and is divided by 7. The CASE statement swaps the remainder with the day of the week. Since the result of the program is assigned to @ANS, this program could be modified and placed in a dictionary record. See the "Interfacing Dictionaries and R/BASIC" in the chapter "Programming with R/BASIC".

Incorrect Use of MOD Function

```
TOT = MOD(49, 0)
```

The second expression uses a 0 (zero).

NEG

The NEG function returns the inverse of a specified number.

NEG(expression)

expression

The specified *expression* must be numeric and must not contain any special characters other than the "-" (minus sign), E scientific notation, and "." (decimal point).

If the *expression* contains characters other than the valid characters, the program will print an error indicating non-numeric data has been intercepted.

Correct Use of NEG Function

```
Y = NEG(X)
```

This returns the inverse of the variable X.

```
BELOW.ZERO = NEG(1)
```

BELOW.ZERO is assigned -1.

```
HUNDRED = NEG(-1E2)
```

100 is assigned to HUNDRED.

```
LOOP
```

```
  PRINT "Input Number " :
```

```
  INPUT X
```

```
UNTIL X = "END"
```

```
  PRINT NEG(X)
```

```
REPEAT
```

```
END
```

The program prints the inverse of the input value

Incorrect Use of NEG Function

```
Q = NEG X
```

Parentheses are missing.

```
A = NEG("ALPHA")
```

Must be numeric value.

NOT

The NOT function compares an expression with the constant 0 (zero) and returns a value of 1 (true) if the given *expression* evaluates to 0, and a value of 0 (false) if the *expression* evaluates to a non-zero quantity.

NOT(expression)

expression

The *expression* may designate any data. The result of NOT is the logical inverse of the specified expression. It generates a value of 1 (one) if the expression evaluates to a 0 (zero), and generates a value of 0 (zero) if the expression evaluates to a non-zero quantity.

For example:

```
A = NOT (0)
```

assigns the value of 1 (one) to the variable identifier A. In making the comparison, 0 (zero) does equal the constant 0 (zero) and the result is therefore true and a 1 (one) is returned.

In this example:

```
A = 243
B = -199
C = NOT (A+B)
```

the variable identifier C is assigned the value 0 (zero) because 44 does not equal the constant 0 (zero) and the expression is therefore false.

In this example:

```
J = 5; K = 3
L = NOT (J OR K)
```

since both are not 0 (zero), the result of NOT is false and the value 0 (zero) is assigned to the variable identifier L.

If the expression evaluates to more than one 0 (zero), the NOT function returns a 0 (false). Spaces or an alphanumeric string will also evaluate to 0 (false).

Correct Use of NOT Function

```
X = 34
Y = 23
PRINT NOT (X AND Y)
```

A value of 0 (zero) will be printed.

```
IF NOT (OK) THEN STOP
```

If the current value of OK is 0 (zero), the program will terminate.

```
R$VAL = NOT (100 > R$)
```

If the value of the expression is 0 (zero), R\$VAL will be assigned 1 (one); otherwise it will be assigned a 0 (zero).

Incorrect Use of NOT Function

```
PRINT NOT X
```

Parentheses are missing.

NULL

The NULL statement is used anywhere in an R/BASIC program when a statement is required but no operation or action is desired.

NULL

NULL

NULL is useful when a statement is required but no operation is desired. For example, consider the following:

```
IF FICA = MAX THEN NULL ELSE GOSUB CALC_FICA
```

If the current value of the variable identifier FICA is equal to the value of MAX, no action is taken and program control proceeds to the next sequential statement. If the value is not equal to MAX, control transfers to a subroutine.

Correct Use of NULL

```
100 NULL
```

At this point in the program, no action will be taken. The statement label (100) may be used as an entry point for a GOTO or GOSUB subroutine.

```
READ CHRQ FROM INV ELSE NULL
```

If no CHRQ exists on the file INV, no action is taken.

```
IF BAL = 0 THEN NULL ELSE
  PRINT "BALANCE DUE"
  GOTO 150
END
```

If the current value of BAL is not zero, the ELSE statement is executed; the program branches to 150. If the balance is zero, control passes to the next sequential statement past the END.

Incorrect Use of NULL

```
IF NULL THEN GOTO 45
```

NULL statement is used as an expression.

```
NULL = A + B
```

NULL is used as a variable identifier.

NUM

The NUM function tests an expression and returns a 1 (true) if the expression is a number or a numeric string. Otherwise it returns a 0 (false).

NUM(expression)

expression

If the *expression* evaluates to a number, a numeric string value, such as scientific notation, or null, NUM returns a value of 1 (true). If the *expression* evaluates to a letter or alphanumeric string, the result is a 0 (false).

In the following example:

```
K1 = NUM(937)
K2 = NUM("673")
K3 = NUM("2G8")
```

the variable identifiers K1 and K2 will each be assigned a value of 1 (true). The variable identifier K3 will be assigned 0 (false).

A decimal point in a numeric string will evaluate to 1 (true). Commas or dollar signs cause even an otherwise all numeric string to evaluate as 0 (false).

ALPHA determines whether the given argument is non-numeric; it is the opposite of NUM.

For more information, see also ALPHA.

Correct Use of NUM Function

```
* EXAMPLE FOR NUM
STRING = ""
LOOP UNTIL STRING = "END"
  PRINT "ENTER DATA ";
  INPUT STRING
  IF NUM(STRING) THEN
    PRINT "DATA ":STRING: " IS NUMERIC "
  ELSE
    PRINT "DATA ":STRING: " IS ALPHA "
  END
REPEAT
END
```

Incorrect Use of NUM Function

```
NUM(144) = GROSS
```

The function appears on the left side of the assignment operator.

```
IF NUM THEN STOP
```

Expression has not been specified.

OCONV

The OCONV function converts data from its internal system format to a corresponding output format.

OCONV(string, conversion)

string

The first expression is the string value that represents the specific data to be converted. The expression is converted to the external format specified by the conversion code defined in the second expression.

The *string* may yield a dynamic array using field marks (ASCII character 254), value marks (ASCII character 253), subvalue marks (ASCII character 252), or text marks (ASCII character 251) as delimiters.

conversion

When the *conversion* code(s) are literal values, they are enclosed in quotes.

Format specifications used in FMT may be used as conversions. See FMT for more information.

For further details on conversion specifications, see

```
OCONV Boolean (B)
OCONV Date (D)
OCONV Datetime (DT)
OCONV Masked Decimal (MD)
OCONV Time (MT)
OCONV MX, HEX, MO, MB
```

Correct Use of OCONV Function

```
TODAY = OCONV (DATE ( ) , 'D2-')
```

Converts today's internal date to output in the form MM-DD-YY.

```
PRINT OCONV (100000, "MD2$")
```

Prints \$1000.00.

```

PRINT @(-1); * clear screen
LOOP
  PRINT 'Enter string to convert : ' :
  INPUT STRING
UNTIL STRING = 'END'
  PRINT 'Enter the conversion code : ' :
  INPUT CODE
  EXT = OCONV (STRING, CODE)
  PRINT EXT: ' = OCONV(' :STRING: ', ' :CODE: ' )'
  INT = ICONV (EXT, CODE)
  PRINT INT: ' = ICONV(' :EXT: ', ' :CODE: ' )'
REPEAT
END

```

The program converts the input string to external format and converts the external format to internal format.

Incorrect Use of OCONV Function

```
X = OCONV (3)
```

A conversion expression is missing.

OCONV Boolean (B)

The OCONV boolean function converts a Boolean true (1) or false (0) into any characters or words.

```
OCONV(string, "B[ {tf}
               [ {true, false} ] ]")
```

string

String can contain any character or word. Any non-zero numbers and any non-null strings are treated as Boolean true. Number zero is treated as Boolean false. A null string is retained as a null string.

B

B indicates a boolean conversion. If no additional options are provided, the conversion assumes the options "Yes,No". In other words, "B" is interpreted the same as "BYes,No".

tf

If the *tf* option is used, a true *string* is converted to the *t* character. Any false *string* is converted to the *f* character.

true,false

If the option *true,false* is used, a true *string* is converted to the *true* word. False *strings* are converted to the *false* word.

Correct Use of OCONV Boolean (B) Specifications

```
TEST = 0
PRINT OCONV (TEST, "B")
```

The program prints "No".

```
CONV = "BError,OK"
OSREAD FILE FROM "C:\AUTOEXEC.BAT"
PRINT OCONV (STATUS ( ) , CONV)
```

The program fragment uses the return value of STATUS() after an OSREAD command to display the results of the operation. Note that due to the return value of STATUS(), the "true" condition is the error condition.

```
GPA = INT ( GPA )
PRINT GPA, OCONV (GPA, "BPassed,Failed")
```

This program fragment uses the integer portion of a student's grade point average to display whether the student has passed or failed.

OCONV Date (D)

The OCONV function with Date (D) conversion specifications converts the data value representing the internal system date to a corresponding output form.

OCONV (expression, "D{y}{c}{E}{F}{G}{H}{I}{J}")

expression

The expression yields a numeric integer.

D

The required D indicates date conversion.

y

The y option indicates the number of digits to be used in the year value output. Zero to four are permitted; zero would indicate no year value wanted; four is actually unnecessary because the default is 4.

c

The c option indicates a character to be used in the output to separate the month, day, and year. When used, it specifies the month value will be output as a two-digit

number with the indicated character separator. The default is the month output as a standard three-letter abbreviation for the month.

E, F, G, H, I, J

The E, F, G, H, I, and J options determine the format for the order in which the day and month appear in the output. For example, an expression value of 7000 produces the following for each format:

Option	Meaning
DE	01 MAR 1987
DF	MAR 1987 01
DG	1987 01 MAR
DH	MAR 01 1987
DI	01 1987 MAR
DJ	1987 MAR 01

- ☛ The internal system date is the number of days since December 31, 1967. This date has been designated internally as day 0 (zero), so all dates after that are represented as the number of days that have elapsed since that specific date. All dates previous to the day 0 (zero) are designated as negative number of days prior to the reference date.

For more information, see DATE(), ICONV Date (D), and TIMEDATE().

Correct Use of OCONV Date (D) Specifications

```
PRINT OCONV(-3477, 'D')
```

Output is 24 JUN 1958

```
PRINT OCONV(-3477, 'D2')
```

Output is 24 JUN 58

```
PRINT OCONV(-3477, 'D4')
```

Output is 24 JUN 1958

```
PRINT OCONV(-3477, 'D/')
```

Output is 06/24/1958

```
PRINT OCONV(-3477, 'D2/')
```

Output is 06/24/58

```
PRINT OCONV(-3477, 'D*')
```

*Output is 06*24*1958*

```
PRINT OCONV(-3477, 'D/E')
```

Output is 24/06/1958

```
PRINT OCONV (-3477, 'D2E')
```

Output is 24 JUN 58

```
PRINT OCONV (-3477, 'D2 J')
```

Output is 58 06 24

```
PRINT OCONV (-3477, 'D2/E')
```

Output is 24/06/58

```
PRINT OCONV (-3477, 'D2/J')
```

Output is 58/06/24

```
A = 5449
```

```
B = "D2/"
```

```
D = OCONV (A, B)
```

D will be assigned 12/01/82.

Incorrect Use of OCONV Date (D) Specifications

```
PRINT OCONV (3477, "D/2")
```

The 2 does not precede the / as required.

```
PRINT OCONV (3477, "DE*")
```

The E is not the last letter as required.

OCONV Datetime (DT)

The OCONV datetime function converts a decimal number into an external date and time specification.

```
OCONV (expression, "DT{dateconv} {^ {n{c}} {timeconv}}")
```

Internal (fractional) Date

The fractional internal date represents the number of days since December 31, 1967 at midnight. As with the "D" conversion, this date has been chosen to represent day and time zero. Dates previous to December 31, 1967 appear as negative numbers.

☛ For more information on the datetime conversion, see ICONV - Datetime (DT).

expression

The expression contains a decimal number with no punctuation other than a period. The whole number portion of the *expression* is assumed to be an internal date. The fractional portion of the *expression* is interpreted as a fractional portion of a day.

If the expression cannot be converted into a datetime format, STATUS() is set to 1.

DT

DT indicates the datetime conversion. If no options are specified, the conversion "DT^1 " is assumed.

dateconv

The *dateconv* can be any valid date conversion as documented under the heading OCONV Date (D).

- ☞ Do not include the "D" in the *dateconv* expression. Include only the optional parameters.

^ (caret)

Use the ^ symbol (caret) to delimit the date conversion specification from the time conversion specification.

n

The *n* specifies the number of characters that will separate the date from the time. The default is one.

c

Use *c* to specify a character used to separate the date from the time. The default is a space. Unless the *timeconv* starts with "H" or "S", *c* is required if *n* is specified,

timeconv

The *timeconv* is any valid time conversion as documented under the topic OCONV Time (MT).

- ☞ Do not include the "MT" in the *timeconv* expression. Include only the optional parameters.

For more information, see DATE(), ICONV Date (D), Time (MT), and TIMEDATE().

Correct Use of OCONV Datetime (DT) Specifications

```
PRINT OCONV(7000.5, 'DT')
```

Output is 01 MAR 1987 12:00

```
PRINT OCONV(7000.5, 'DT2^HS ')
```

Output is 01 MAR 87 12 00 00PM

```
PRINT OCONV(7000.5, 'DT^3*/')
```

*Output is 01 MAR 1987***12/00*

```
PRINT OCONV(7000.5, 'DT/^S')
```

Output is 03/01/1987 12:00:00

```
PRINT OCONV(7000.5, 'DT:^3H')
```

Output is 03:01:1987 12:00PM

```
PRINT OCONV(-3966.25, 'DT/^3H')
```

Output is 02/19/1957 06:00PM

```
PRINT OCONV(-3966.75, 'DT2/^H')
```

Output is 02/19/57 06:00AM

Incorrect Use of OCONV Date (D) Specifications

```
PRINT OCONV(7000.5, "DTD2/")
```

The date conversion includes a "D".

```
PRINT OCONV(7,000.5, "DT")
```

The expression includes punctuation other than a decimal point.

OCONV MX, HEX, MO, MB

The OCONV function with MX, HEX, MO, or MB conversion specifications converts values from decimal formats to hexadecimal, octal, and binary formats.

```
OCONV(expression, "MX")  
OCONV(expression, "HEX")  
OCONV(expression, "MO")  
OCONV(expression, "MB")
```

expression

The expression yields a decimal value. Scientific notation is accepted.

MX

In MX, the expression must be a decimal number (base 10) and is converted to the equivalent hexadecimal character set (base 16).

HEX

In HEX, the expression must be a string of ASCII characters with a maximum length of 32,766 and is converted into a string of hexadecimal character sets.

MO

In MO, the expression must be a decimal number and is converted to an ASCII representation of an octal (base 8) number.

MB

In MB, the expression must be a decimal number and is converted to an ASCII representation of a binary (base 2) number.

If expression yields an incorrect value a 0 (zero) will be returned.

- ✎ An understanding of different number bases is required for the effective use of these specifications.

Use the following table as a guide to conversions to different number bases:

Conversion Code	ICONV	OCONV
MX	hexadecimal to decimal	decimal to hexadecimal
MO	octal to decimal	decimal to octal
MB	binary to decimal	decimal to binary

See ICONV MX, HEX, MO, MB for more information.

Correct Use of OCONV MX, HEX, MO, MB Specifications

A = 19
B = "MX"
C = OCONV (A, B)
The value 13 is assigned to the variable identifier C.

A = "A"
B = "HEX"
C = OCONV (A, B)
The value 41 is assigned to the variable identifier C.

A = 1E2
B = "MB"
C = OCONV (A, B)
The value 1100100 will be assigned to the variable identifier C.

Incorrect Use of OCONV MX, HEX, MO, MB Specifications

PRINT OCONV (A, MX2)
2 is not a valid specification.

OCNV Masked Decimal (MD)

The OCNV function with Masked Decimal (MD) conversion specifications converts the value representing the internal system decimal data into a corresponding output form.

OCNV(string, "MDn{f}{,}{\$}{- or < or C or D}{P}{Z}{T}{xc}")

MD

MD indicates Masked Decimal conversion.

- ✎ The MC (Masked Comma) alternative to MD uses a “,” (comma) to indicate decimal places. Thousands are masked with a “.” (period).

n

A single digit specifies number of digits to right of decimal point; the values A - H are used for 10 - 18. A 0 (zero) means no decimal point.

f

In the *f* option, a single digit (0-9) specifies the scaling factor, in other words, the position of the implied decimal point in the data. The string is divided by 10 to a factor of *f*. This is equivalent to moving the decimal point one place to the left. When ICONV is used, the string is multiplied by 10 to a factor of *f*. If *f* is not specified, the default is the same as *n*. If the P option is specified, *f* will not be used.

- ✎ The data value is rounded if *n* is less than *f*. Data that is input with a null value is output as null. Invalid input data will be returned by the function.

,

The , option inserts commas to separate thousands.

\$

The \$ option outputs a \$ (dollar sign) prefix.

-

The - option specifies a - (minus sign) suffix for negative data. Positive data output is suffixed with a single space.

<

The < option specifies < > (surrounding brackets) for negative data values. Positive data is suffixed with a single space.

C

The C option specifies a suffix CR to indicate CREDIT instead of a minus sign. The positive data output has two spaces suffixed.

D

The D option is similar to C but suffixed with DB to indicate DEBIT.

P

The P option specifies that no scaling should be performed if input data contains a decimal point.

Z

The Z option outputs a data value of 0 (zero) as null.

T

The T option truncates data rather than rounding.

xc

The xc option marks space reserved for data where x is the number of spaces to mark and c is the character to use in marking.

Correct Use of OCONV Masked Decimal (MD) Specifications

```
PRINT OCONV(98765, "MD0, ")
```

Output is 98,765

```
PRINT OCONV(98765, "MD2")
```

Output is 987.65

```
PRINT OCONV(98765, "MD2$")
```

Output is \$987.65

```
PRINT OCONV(-817263, "MD2$<")
```

Output is <\$8172.63>

```
PRINT OCONV(-817263, "MD1, -")
```

81,726.3-

```
PRINT CONV(213579, "MD2$, 10*")
```

*Output is \$*2,135.79*

```
PRINT OCONV(1234, "MD2, 10*")
```

*Output is *****12.34*

```
PRINT OCONV(-817263, "MC1. ")
```

Output is -81.726,3

OCONV Masked Scientific (MS)

The OCONV function with Masked Scientific (MS) output conversion specifications converts internal system representation to a corresponding scientific notation.

```
OCONV(string, "MS{e {mm} }")
```

string

The *string* yields a numeric value that may include decimal places. If the string yields invalid data, a 0 (zero) will be returned.

MS

The MS specifies scientific conversion.

e

The *e* option specifies the number of digits allocated for the power of ten to which the number is raised. The default is 2 (two).

mm

If the *e* option is specified, the *mm* option specifies the mantissa, which is the number of places following the decimal point. The default is six places. The hardware limit for the mantissa is 13. The mantissa may be one or two digits.

For more information, see ICONV Masked Scientific (MS).

Correct Use of OCONV Masked Scientific (MS) Specifications

```
PRINT OCONV(1.23, "MS")
```

Output is 1.230000E+00

```
PRINT OCONV(-123, "MS22")
```

Output is -1.23E+02

```
PRINT OCONV(123, "MS15")
```

Output is 1.23000E+2


```
PRINT OCONV(123456, "MS23")
```

Output is 1.235E+05

Incorrect Use of OCONV Masked Scientific (MS) Specifications

```
X = OCONV(1.345)
```

Specifications are missing.

```
OCONV(Y) = 2
```

The function appears on the left side of the assignment operator.

OCONV Time (MT)

The OCONV function with Time (MT) conversion specifications converts the data values representing internal system time to a corresponding output form.

```
OCONV(expression, "MT{H}{S}{c}")
```

expression

The *expression* yields a numeric integer.

MT

The MT specifies time conversion.

H

The H option specifies 12-hour format with AM and PM indicated in the result. The default is a 24-hour clock.

S

The S option specifies that seconds are to be used in the output.

c

The c option specifies the character that is to be used to separate hours, minutes, and seconds. Colons are the default.

Internal system time is calculated as the number of seconds past midnight. Since a 24-hour day has 86,400 seconds, the string value has to evaluate to less than 86,400 to be a valid expression.

For more information, see ICONV Time (MT) and TIME().

Correct Use of OCONV Time (MT) Specifications

```
PRINT OCONV(43260, "MT")
```

Output is 12:01

```
PRINT OCONV(43260, "MTH")
```

Output is 12:01PM

```
PRINT OCONV(43260, "MTS")
```

Output is 12:01:00

```
PRINT OCONV(43260, "MTHS")
```

Output is 12:01:00PM

```
PRINT OCONV(61200, "MT")
```

Output is 17:00

```
PRINT OCONV(61200, "MTH")
```

Output is 05:00PM

```
PRINT OCONV(61200, "MTS")
```

Output is 17:00:00

```
PRINT OCONV(61200, "MTHS")
```

Output is 05:00:00PM

```
A = 73680
```

```
B = "MTH"
```

```
T = OCONV(A, B)
```

T is assigned the value 08:28PM

Incorrect Use of OCONV Time (MT) Specifications

```
T = OCONV("654")
```

Conversion specification is missing.

```
T = OCONV(43260, "MH")
```

Illegal specification is used.

ON GOSUB

The ON GOSUB statement transfers program control to one of several internal subroutines that are listed as statement labels. Selection of the specific subroutine is determined by computing the current value of an index expression.

```
ON index GOSUB label{:} [{,label{:})::]
```

index

When the ON GOSUB statement is encountered, the current value of the *index* is determined, and the program transfers control to the proper subroutine.

If the *index* evaluates to 1 (one), control will branch to the first subroutine designated; if the expression evaluates to 2 (two) the program will branch to the second subroutine; and so forth.

The *index* is evaluated and truncated to an integer value. If the resulting value is less than 1 (one), or if the value exceeds the number of subroutine labels, an error results.

label

The *label* or *labels* that are designated indicate subroutines within the program. Either numeric or alphanumeric labels are valid; colons may be used to increase clarity and to differentiate the *labels* from possible variable identifiers.

The ON GOSUB statement must be written on one line with the labels separated by commas.

Program control will return to the main program sequence when RETURN or RETURN TO is encountered in the subroutine.

For more information, see RETURN and RETURN TO.

Correct Use of ON GOSUB

```
SCR.LOC1 = @(20,12)
PRINT @(-1)
SCREEN = @(10,10) : "ENTER 1, 2, 3, OR 4 TO END"
PRINT SCREEN :
LOOP
    INPUT ANS,1
    ON ANS GOSUB LABEL1, LABEL2, LABEL3, LABEL4
REPEAT
*
LABEL1:
PRINT SCR.LOC1 : "You pressed a 1 " : SCREEN :
RETURN
LABEL2:
PRINT SCR.LOC1 : "You pressed a 2 " : SCREEN :
RETURN
LABEL3:
PRINT SCR.LOC1 : "You pressed a 3 " : SCREEN :
RETURN
LABEL4:
STOP
```

Incorrect Use of ON GOSUB

```
ON GOSUB 350, 400
```

The index expression is missing.

```
ON PL# > GOSUB A:,B
```

An expression is used that will not yield all possible index values.

ON GOTO

The ON GOTO statement transfers program control to one of several statement labels that are listed. Selection of the specific label is determined by computing the current value of an index expression.

```
ON index GOTO label{:} [{,label{:}}::]
```

index

When the ON GOTO statement is encountered, the current value of the *index* is determined and the program transfers control to the proper label.

If the *index* evaluates to 1 (one), control will branch to the first statement *label*; if the expression evaluates to 2 (two), the program will branch to the second statement *label*, and so on.

The *index* is first evaluated and then the result is truncated to an integer value. If the resulting value is less than 1 (one) or if the value exceeds the number of statement *labels*, an error message is generated.

label

The *labels* that are designated in the list may either precede or follow the ON GOTO statement, but they must be valid labels that are defined in the program. If the specified label does not occur in the program, a compiler error message will result.

Numeric or alphanumeric labels may be used. A colon (:) may be used to increase clarity and to differentiate them from possible variable identifiers.

The ON GOTO statement must be written on one line with the statement labels separated by , (commas) as in the following example:

```
ON T GOTO 200, 250, 375
```

Use of the word TO in GOTO is optional.

For more information, see ON GOSUB.

Correct Use of ON GOTO

```
ON X+Y GOTO 30, 9, 44, 21
```

The value of the expression X+Y will determine where the program will be transferred. A value of 1 (one) will transfer control to label 30, a value of 2 to label 9, and so on.

Incorrect Use of ON GOTO

```
ON C-2=1 GOTO 24,52,6
```

An unconditional branch to label 24 would result because the expression is a logical quantity and can have only 1 (one) or 0 (zero) as a value.

OPEN

The OPEN statement prepares an Advanced Revelation file for subsequent reading, writing, deletion, and locking of records. A file cannot be accessed by a READ or WRITE statement unless you have previously used OPEN.

Each file must be opened with a separate OPEN statement. Any number of files may be opened at any point in the program.

Files opened with the OPEN command need not and cannot be closed.

```
OPEN expression TO filevar THEN|ELSE statements
```

expression

The *expression* designates the Advanced Revelation file that is to be selected by OPEN. To open the dictionary, use DICT as the first part of the filename (example: "DICT CUSTOMERS"), and to open the data portion of the file, use only the filename (example: "CUSTOMERS").

filevar

After a successful OPEN operation, the *filevar* contains information used in subsequent reference to the file.

THEN

The *statement(s)* following THEN are executed if a file is opened successfully.

☛ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the file cannot be opened. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

Correct Use of OPEN

```

EQU BEEP TO CHAR(7)
OPEN "CUSTOMERS" TO CUST.FILE THEN
  OPEN "DICT CUSTOMERS" TO @DICT ELSE
    STOP "Dictionary of CUSTOMERS not available!"
  END
END ELSE
  FSMG()
  STOP
END
DONE=0
LOOP
  PRINT "Enter a record key:":
  INPUT @ID
  IF @ID EQ "END" THEN DONE = 1
UNTIL DONE DO
  READ @RECORD FROM CUST.FILE,@ID THEN
    PRINT {COMPANY} , {CITY} , {ST}
  END ELSE
    PRINT BEEP : @ID : " not available in CUSTOMERS"
  END
REPEAT
STOP

```

Both dictionary and data portions of the CUSTOMERS file are opened. The dictionary records COMPANY, CITY, and ST are used to print fields.

Incorrect Use of OPEN

```
OPEN "" TO RM ELSE STOP
```

The file name is missing.

```
OPEN ABC TO ZEN
```

An ELSE or THEN clause is required.

OSBREAD

The OSBREAD statement allows R/BASIC to read operating system files larger than 64K by specifying a beginning byte position and a length less than 64K. OSBREAD functions only with operating system files that have been previously opened with an OSOPEN file.

OSBREAD variable FROM filevar AT byte LENGTH length

variable

The variable into which the file portion will be read.

filevar

The variable to which the operating system file was assigned when it was opened with an OSOPEN statement.

byte

Byte is an integer identifying how many bytes (offset) into the operating system file that the read operation should begin. A 0 (zero) integer indicates the read should start at the beginning of the file.

length

The *length* is an integer that tells how many bytes to read. This number cannot exceed 65,530.

STATUS()

After the execution of an OSBREAD statement, the STATUS() of the read is returned, with one of the following codes:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied by operating system
4	File does not exist
5	Undefined error
7	Invalid beginning byte position

- ☛ For more information, see OSBWRITE and OSOPEN. For more information on operating system file access, see "Operating System File Access" in the chapter "Programming with R/BASIC".

Correct Use of OSBREAD

```

CRLF = CHAR(13) : CHAR(10)
TOP:
OSOPEN "OSTEST" TO FILE.VAR ELSE
  PRINT "Creating operating system file" : CRLF
  OSWRITE "" TO "OSTEST"
  GO TOP
END
PRINT "STATUS is " : STATUS()
REVREC1 = "Field 1 REC 1":@FM:"Field 2 REC 1":@FM:"Field 3 REC 1"
REVREC2 = "Field 1 REC 2":@FM:"Field 2 REC 2":@FM:"Field 3 REC 2"
FILE = REVREC1 : CRLF : REVREC2
SWAP @FM WITH ", " IN FILE
OSWRITE FILE TO FILE.VAR
IF STATUS() = 0 THEN
  PRINT CRLF : "OSWRITE results follow:"
  PERFORM "PC TYPE OSTEST"
END
OSREAD NEWFILE FROM FILE.VAR THEN

```

```

    PRINT CRLF : "OSREAD results" : CRLF : NEWFILE
END
PRINT CRLF : "OSBREAD results follow"
FILE.LEN = LEN(NEWFILE)
FOR X = 0 TO FILE.LEN
    OSBREAD BYTE FROM FILE.VAR AT X LENGTH 1
    PRINT BYTE : "-" :
    OSBWRITE NEWFILE[-X,1] TO FILE.VAR AT X
NEXT X
PRINT CRLF : "OSBWRITE result follows" :
PCPERFORM "TYPE OSTEEST"
OSCLOSE FILE.VAR
IF NOT( STATUS() ) THEN PRINT CRLF : CRLF : "OSTEEST closed"
OSDELETE "OSTEEST"
IF NOT( STATUS() ) THEN PRINT CRLF : "Deleted" ELSE
    PRINT CRLF : "Not deleted"
END

```

The program shows a number of methods for reading and writing operating system files and may be referred to for other OS statements.

OSBWRITE

The OSBWRITE statement writes data into a previously opened operating system file beginning at a specified point in the file. OSBWRITE functions only with operating system files that have been previously opened with an OSOPEN statement.

OSBWRITE expression ON TO filevar AT byte
--

expression

Identifies the data to be written to the operating system file.

filevar

The variable to which the operating system file was assigned when it was opened with an OSOPEN statement.

byte

Byte is an integer that specifies the position in the operating system file where the write will begin. A 0 (zero) indicates the write should start at the beginning of the file.

Note that OSBWRITE has no length. The number of bytes written is the same as the number of bytes in *expression*.

STATUS()

After the execution of an OSBWRITE statement, the STATUS() of the write is returned, with one of the following codes:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied by operating system
3	Disk or directory full
4	File does not exist
5	Undefined error
6	Attempt to write a read only file
7	Invalid beginning byte position

- For more information, see the OSBREAD, OSCLOSE, and OSOPEN statements. For more information on operating system file access, see "Operating System File Access" in the chapter "Programming with R/BASIC".

Correct Use of OSBWRITE

```
OSOPEN "TEST" TO FILE.T ELSE STOP
OSBWRITE ABC ON FILE.T AT 0
IF STATUS( ) THEN GOSUB ERROR:
```

Opens operating system file TEST and assigns to variable FILE.T. Writes the expression ABC to FILE.T beginning at byte 0 of the file. If the status is not 0 (zero), the program will branch to subroutine ERROR: . See also the example under OSBREAD.

Incorrect Use of OSBWRITE

```
OSBWRITE ABC ON FILE.T
```

No indication of where to begin writing the expression.

OSCLOSE

The OSCLOSE statement closes an operating system file previously opened with an OSOPEN statement. A file that has been opened by OSOPEN must be closed. This differs from an Advanced Revelation file, which cannot be closed.

OSCLOSE filevar

filevar

The variable to which the operating system file was assigned when it was opened with an OSOPEN statement.

STATUS()

After the execution of an OSCLOSE statement, the STATUS() of the close is returned, with one of the following codes:

Value	Meaning
0	no error
5	undefined error

- ☛ For more information, see OPEN and OSOPEN. For more information on operating system file access, see “Operating System File Access” in the chapter “Programming with R/BASIC”.

Correct Use of OSCLOSE

```
OSOPEN "TEST" TO FILE.T ELSE STOP
OSCLOSE FILE.T
IF STATUS( ) THEN GOSUB ERROR:
```

Opens the operating system file TEST and assigns it to variable FILE.T. Closes the operating system file associated with variable FILE.T. If STATUS() does not return a 0 (zero), the program will branch to subroutine ERROR: . See also the example under OSBREAD.

OSDELETE

The OSDELETE statement deletes an operating system file.

OSDELETE file

file

The name of the operating system file (including drive or directory path, filename, and extension) to be deleted.

STATUS()

After the execution of an OSDELETE statement, the STATUS() of the delete is returned, with one of the following codes:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied by operating system
4	File does not exist
5	Undefined error

- ☛ Files to be deleted should not be open. See OSCLOSE. For more information on operating system file access, see “Operating System File Access” in the chapter “Programming with R/BASIC”.

Correct Use of OSDELETE

```
OSDELETE "A:TEST.DOC"
```

Deletes the operating system file called TEST.DOC on drive A. See also the example under OSBREAD.

Incorrect Use of OSDELETE

```
OSOPEN "TEST" TO FILE.T ELSE STOP  
OSDELETE FILE.T
```

The OSDELETE function requires an operating system file name, not an opened file variable.

OSOPEN

The OSOPEN statement opens an operating system file for use with OSBREAD and OSBWRITE. OSBREAD and OSBWRITE can access operating system files only after being opened with an OSOPEN statement. Files opened by OSOPEN should be closed with OSCLOSE.

OSOPEN file TO filevar THEN|ELSE statements

file

The name of the operating system file (including drive or directory path, filename, and extension) to be opened.

filevar

Identifies the variable to which the file is assigned.

THEN

THEN may be used to define a clause of *statements* to be executed when the OSOPEN statement is successful.

☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The ELSE clause is executed if the OSOPEN is unsuccessful.

STATUS()

After the execution of an OSOPEN statement, the STATUS() of the open is returned, with one of the following codes:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied by operating system
4	File does not exist
5	Undefined error

☛ For more information, see OSBREAD, OSCLOSE, and OSBWRITE. For more information on operating system file access, see “Operating System File Access” in the chapter “Programming with R/BASIC”.

Correct Use of OSOPEN

```
OSOPEN "TEST" TO FILE.T ELSE
  GOSUB STATUS.CHECK
END
```

Opens operating system file TEST to variable FILE.T. See also the example under OSBREAD.

Incorrect Use of OSOPEN

```
OSOPEN "TEST" ELSE STOP
Not assigned to a variable.
```

OSREAD

The OSREAD statement reads data from an operating system file and assigns that data to a variable in an R/BASIC program.

```
OSREAD var FROM file THEN|ELSE statements
```

OSREAD

OSREAD assigns an entire operating system file to the variable, where OSBREAD assigns a portion of an operating system file to its variable. In both cases, the variable may contain a maximum of 65,530 bytes.

var

The variable into which the file will be read.

file

The *file* yields the name of the valid operating system file, including the drive or directory path, the filename, and extension. This is not a file variable. The operating system file is assumed, so no file needs to be opened.

THEN

THEN may be used to define a clause of statements to be executed when the OSREAD statement is successful.

☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The ELSE clause is executed when the OSREAD is unsuccessful.

STATUS()

After the execution of an OSREAD statement, the STATUS() function returns a code indicating the success or failure of the operation. Possible values for STATUS() are:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied by operating system
4	File does not exist
5	Undefined error

☞ For more information on operating system file access, see "Operating System File Access" in the chapter "Programming with R/BASIC". See also OSWRITE.

Correct Use of OSREAD

```
OSREAD FILE.T FROM "A:TEST.DOC" THEN
  IF STATUS( ) THEN GOSUB ERROR
END ELSE
  STOP
END
```

The program opens operating system file TEST.DOC to a variable named FILE.T. See also the example under OSBREAD.

Incorrect Use of OSREAD

```
OSREAD FROM FILE.T ELSE STOP
```

No variable identified to read file into.

OSWRITE

The OSWRITE statement writes data to a specified operating system file.

`OSWRITE expression

ON
TO

 file`

expression

The *expression* yields the string or dynamic array to be written.

file

The *file* is the name of the operating system file (including the drive or directory path, the filename, and extension) the expression is to be written to. This is not a file variable. The operating system file is assumed, so no file needs to be opened.

If the operating system file exists, it will be overwritten with the new data. If the operating system file does not exist, it will be created.

STATUS()

After the execution of an OSWRITE statement, the STATUS() of the write is returned, with one of the following codes:

Value	Meaning
0	No error
1	Bad operating system filename
2	Access denied
3	Disk or directory full
5	Operating system error not defined elsewhere
6	Attempt to write a read only file

☞ For more information see OSREAD. For more information on operating system file access, see “Operating System File Access” in the chapter “Programming with R/BASIC”.

Correct Use of OSWRITE

`OSWRITE ABC TO "A:TEST.DOC"`

Writes the expression ABC into the operating system file named TEST.DOC on drive A. See also the example under OSBREAD.

Incorrect Use of OSWRITE

`OSWRITE TO "TEST"`

No expression identified to write to the operating system file.

OUT

The OUT statement is used to output a single value to any port.

✎ OUT is not currently implemented for OS/2.

OUT *port*, *out*

port

The *port* must evaluate to a decimal value between 0 and 65,535. This number represents the port that is to receive the output.

out

The *out* is the value that is to be output. It must evaluate to a decimal number between 0 and 255.

The OUT statement is often used in conjunction with the INP function. The OUT and INP statements are used for direct memory addressing. These statements may be used to manipulate hardware configurations.

Warning ! The OUT and INP commands require knowledge of direct memory addressing and hardware. Uninformed use may cause serious problems.

✎ For more information, see the BIT functions and INP.

Correct Use of OUT

```
PORT = 1305
OUT PORT, 27 ; *OUTPUT ESCAPE TO PORT 1305
OUT PORT, 48 ; * OUTPUT ASCII ZERO TO PORT 1305
FOR C = 65 TO 70
    OUT PORT, C ; *OUTPUT "ABCDEF" TO PORT 1305
NEXT C
```

Incorrect Use of OUT

```
X = OUT(PORT, 3)
```

OUT is not a function.

```
OUT
```

Expressions are missing.

PAGE

The PAGE statement provides the user with a method of forcing the current output device (screen or printer) to advance one page.

PAGE

Using PAGE

PAGE causes the current output device to print any specified FOOTING and HEADING statements when moving to the new PAGE.

PAGE will use the current output device set by the PRINTER statement. The default output device is the monitor.

☛ For more information, see FOOTING, HEADING, and PRINTER.

Correct Use of PAGE

```
FOR I = 1 TO 100
  IF NOT (MOD (I,5)) THEN PAGE
  PRINT I, MOD (I,5)
NEXT
```

The current output device moves to the top of the next page at every fifth line.

Incorrect Use of PAGE

```
PAGE 5
5 not needed.
```

PCPERFORM

The PCPERFORM statement allows the R/BASIC program to use operating system commands.

PCPERFORM expression

Using PCPERFORM

The PCPERFORM statement allows the use of any operating system command in an R/BASIC program. The expression must yield the operating system command that is to be performed.

expression

The *expression* is executed as though it had been typed at the operating system default drive prompt. The operating system command processor must be accessible

to the operating system for PCPERFORM to work. Under MS-DOS and OS/2, the environmental variable COMSPEC must be set.

- ☞ PCPERFORM and SUSPEND function identically in OS/2 versions.

When Advanced Revelation is re-entered after performing operating system commands the path is reset to its status before the execution of the PCPERFORM.

- ☞ For more information, see the PC, ROLLOUT, and SUSPEND commands in the "TCL Command Reference" chapter of *TCL*, and your operating system manual.
- ☞ Do not execute any programs that will remain resident in memory upon termination. These may interfere with Advanced Revelation.

Correct use of PCPERFORM

```
CMD = "DIR"
LOOP
  PCPERFORM CMD
  PRINT "ENTER OS COMMAND":
  INPUT CMD
UNTIL CMD = "END"; REPEAT
```

The program will perform operating system commands until END is typed.

Incorrect Use of PCPERFORM

```
PCPERFORM
PCPERFORM requires an operating system command.
```

```
PC PERFORM
No space allowed after PC.
```

PERFORM

The PERFORM statement allows a program to send a command to TCL for execution and then return to the calling program.

PERFORM command

command

The *command* is a character string that must contain the exact command as it would be typed at TCL.

When the PERFORM statement is executed, a recursive call to the TCL processor is made, the *command* is performed, and control is returned to the next statement following the PERFORM statement.

Use this statement to create batch-like procedures. For example:

```
PERFORM "SELECT DOS WITH SIZE < 10000"  
PERFORM "EDIT DOS"
```

These two PERFORM statements would select the DOS file and then call the editor with the selected list. When the edit is terminated, control would return to the program.

PERFORM versus EXECUTE

Use the EXECUTE statement to suspend all current processing to execute another subsystem or process. Unlike the EXECUTE statement, PERFORM passes the following information to the new level of TCL:

- The currently active list of keys
- The contents of @DATA assigned by the most recent DATA statement
- The user-defined variables: @RECUR0 through @RECUR4
- The printer status
- The BREAK status
- The ECHO status

Correct Use of PERFORM

```
PERFORM "SELECT CST WITH A ZIP OF 98333"  
PERFORM "LIST CST COMPANY ADDRESS CITY ZIP"
```

The customer file is selected. The selected list of keys is used by the LIST statement.

```
PERFORM ""
```

The next level is entered.

```
PERFORM "MAIN.MENU"
```

The VOC item MAIN.MENU is performed.

Incorrect Use of PERFORM

```
PERFORM
```

The expression is missing.

```
PERFORM X, Y
```

Two expressions are used where only one is allowed.

PRINT

The PRINT statement outputs data to the system printer or to the terminal screen.

```
PRINT {expression [{,expression}::]} {::}
```

PRINT

The PRINT statement writes values to an output device, a terminal screen, or a printer.

Output is toggled between the screen and the printer using the PRINTER statement. Output can be redirected to a disk file using the TCL PDISK command. For more information, see PDISK in *TCL*.

- ☞ When using the PRINT statement to output to the screen in Advanced Revelation's windowing environment, you should catalog or run the program with the (B) option to preserve and restore the window environment properly. An alternative to PRINT is the system subroutine MSG. See MSG in the "System Subroutines" chapter for more information, and CATALOG and RUN in *TCL* for more information on (B).

expression

The *expression(s)* may be any valid R/BASIC expressions.

The printed output may be formatted into print zones by placing commas between the *expressions*. A comma following an expression indicates that the contents of the next expression is to be printed in the next zone. Each print zone consists of 16 spaces.

If an expression is not given in the PRINT statement, a blank line will be output.

Logical and comparison expressions must be resolved before printing. To do this enclose them in parentheses. For example:

```
PRINT (X+Y)
```

: (colon)

A comma or colon (:) at the end of the line in a PRINT statement indicates that the cursor is not to go to a new line. Output from the next executed PRINT statement will be printed on the same line.

- ☞ For more information, see @, FMT, and PRINTER.

Correct Use of PRINT

```
PRINT EXTRACT (B, 12, 3, 0)
```

Prints value 3 of field 12 in dynamic array B.

```
PRINT "at " :TIMEDATE()
```

Prints "at" with the value of TIMEDATE concatenated.

```
PRINT INT(65.6)
```

Prints the integer 65.

```
READ C FROM TEST2, KEY ELSE  
  PRINT "Unable to read ":KEY  
END
```

A message is printed if KEY cannot be read.

Incorrect Use of PRINT

```
PRINT QUANT COST
```

Expressions are not separated by either a comma or a colon.

```
PRINT BALANCE DUE:BAL
```

The literal string BALANCE DUE is not enclosed in quote marks.

PRINTER

The PRINTER statement provides you with the option of directing the output of a PRINT statement to the printer or the screen. The default output unit is the user's display screen.

PRINTER ON OFF expression
--

Using PRINTER

The output of PRINT statements will appear on the user's display screen unless directed otherwise by a PRINTER statement. The @ function (cursor location) should not be used following the execution of a PRINTER ON statement.

If you have directed output to a printer, the R/BASIC PRINT statement will attempt to verify that the printer is available before printing. This test can be skipped by setting the variable @PDISK_ON to true.

ON

After a PRINTER ON statement, all subsequent output from the PRINT statement is routed to the current printer device.

OFF

A PRINTER OFF statement causes output to be redirected to the screen. The printer will also be turned off when the first PERFORM statement is encountered, or at program termination.

expression

If the *expression* evaluates to true (non-zero), PRINTER ON is set. If the value of the expression is false (zero) or null, PRINTER OFF is set.

☞ For more information, see GETPRINTER.

Correct Use of PRINTER

```
PRINTER ON
PRINT OCONV (DATE ( ) , "D2")
```

The printer will print the current system date in the D2 format.

```
PRINTER OFF
PRINT "XYZ"
```

The terminal will show XYZ.

Incorrect Use of PRINTER

```
PRINTER
```

ON or OFF is not used.

```
PRINTER X*Y
```

*A logical expression, X*Y, is used instead of ON or OFF.*

PROMPT

The PROMPT statement designates the character that will be displayed when an INPUT statement is encountered.

PROMPT expression

expression

The *expression* designates the character that is to serve as the prompt on the terminal screen. It must evaluate to a single character. Only the first character will be used if more than one is specified.

When the prompt character appears on the screen, the system is ready to accept data from the terminal operator.

If a PROMPT statement is not executed, the default character is a question mark (?).

If the input data is supplied from a DATA statement instead of from the keyboard, the data will be displayed on the screen after the prompt character.

☞ For more information, see INPUT.

Correct Use of PROMPT

PROMPT "="

The = (equal sign) will appear on the screen as a prompt character.

PROMPT "@"

An @ (at-sign) will appear on the screen as a prompt character.

PROMPT A

Current value of A will appear on the screen as a prompt character.

PROMPT ""

No prompt character will be displayed.

Incorrect Use of PROMPT

PROMPT *

Quotation marks are not used with a literal character.

PROMPT

Expression is missing.

PUT.CURSOR

PUT.CURSOR changes the current video cursor position and shape.

 The word "cursor" here refers to the blinking line or box that indicates your position on the video screen.

PUT.CURSOR(variable)

variable

PUT.CURSOR requires a four-byte argument. Each byte represents one attribute to be set for the cursor. A value is established by passing the ASCII character whose sequence number corresponds to the desired value.

The four bytes in *variable* are (left to right):

- 1 the x (column) position (0-79) of the cursor
- 2 the y (row) position (0-24) of the cursor
- 3 the bottom scan line (0-13) for the cursor shape
- 4 the top scan line (0-13) for the cursor shape

The cursor scan line consists of 14 lines, numbered 0 through 13. Line 0 is the top of the cursor, 13 the bottom. Cursor height is determined by establishing a starting

and ending scan line. For information on returning the current cursor information, see GET.CURSOR.

☞ Cursor blink is a function of the hardware, and cannot be changed.

Correct Use of PUT.CURSOR

```
OPEN "CUSTOMER" TO CUSTOMER_FILE ELSE
  * not attached
  CALL MSG("201", "", "", "CUSTOMER")
END
GET.CURSOR ( CURSOR_INFO ) ; * fetch current info
@ID = ""
CALL MSG("Enter record key:", "R", @ID, "")
READ @RECORD FROM CUSTOMER_FILE, @ID THEN
  BOT = CHAR(13) ; * existing record
  TOP = CHAR(6)
END ELSE
  BOT = CHAR(13) ; * new record
  TOP = CHAR(0)
END
CURSOR_INFO[3,1] = BOT
CURSOR_INFO[4,1] = TOP
* set cursor shape (position not changed)
PUT.CURSOR( CURSOR.INFO )
GOSUB PROCESS
STOP
```

The program prompts for a record id. If the record is new, the cursor is set to its full height. If the record exists, the cursor is set to half height. Note: the local subroutine PROCESS is not shown.

PWR

The PWR (power) function calculates the value of an expression when it is raised to the power designated by the second expression.

PWR(expression, power)

expression

The *expression* yields a numeric value.

power

The PWR function raises the value of the first *expression* to the *power* designated in the *power* and returns that value. For example:

```
PRINT PWR(2, 5)
```

raises the number 2 to the fifth power and prints the value 32.

If the *power* is 0 (zero), a 1 (one) is returned. If the first *expression* is 0 (zero) and the power is any number except 0 (zero), the PWR function returns a value of 0 (zero).

- ✎ Negative values can be raised to either positive or negative powers. However, a warning message will occur if the attempt is made to raise a negative value to a non-integer power.

Correct Use of PWR Function

```
PRINT PWR(3,3)
```

Prints 27.

```
PRINT PWR(X,0)
```

Prints a 1 (one) if the value of X is not 0 (zero).

```
WORDMAX = PWR(2,16)-1
```

Sets variable identifier WORDMAX to 65535.

Incorrect Use of PWR Function

```
X = PWR 3,2
```

Parentheses are missing.

```
PRINT PWR(5)
```

The second expression is missing.

```
Z = PWR (2,5)
```

A space is used between the function and the parenthesis.

QUOTE

The QUOTE function places double quotation marks (") around a given expression.

QUOTE (expression)

expression

The *expression* can also be enclosed in double quote marks (") by using other R/BASIC methods, but the results are more confusing. For instance, the following two lines of code are equivalent:

```
SEL = 'SELECT CM WITH DATE " ' :DT: ' " '
SEL = "SELECT CM WITH DATE " :QUOTE (DT)
```


Correct Use of QUOTE Function

```
PRINT "THE VALUE " : QUOTE(X) : " IS OUT OF RANGE "
```

```
PERFORM "SELECT CUSTS IF TYPE MATCHES " : QUOTE(Z)
```

The values contained in the variables X and Z are surrounded in quote marks.

Incorrect Use of QUOTE Function

```
QUOTE Z
```

Parentheses are missing.

```
Y = QUOTE (A,B,C)
```

Uses more than one expression.

READ

The READ statement reads a record from a file and assigns its value to a variable. The record that is read cannot be greater than 65,530 characters in length.

```
READ variable FROM [filevar  
                    [CURSOR cursorvar]] ,key THEN|ELSE statements
```

variable

Variable is assigned the value of the data that is read from the record specified by the *key*.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement and must be preceded with the word CURSOR.

If the file being accessed has had control features added, cursor access will automatically invoke domain conversion during a READ.

- ☞ For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

filevar

Filevar is a file variable created by a previous OPEN statement.

key

The record referenced by *key* will be read from the file identified by *filevar* or the file accessed using the cursor in *cursorvar*.

THEN

The *statement(s)* following THEN are executed if a record is read successfully.

- ☛ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the record in *variable* cannot be read. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains details about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

- ☛ For more information, see OPEN, READO, READV, MATREAD and WRITE. See also "Dynamic String Arrays" in "Programming with R/BASIC".

Correct Use of READ

```
DECLARE SUBROUTINE FMSG
EQU TRUE$ TO 1
EQU FALSE$ TO 0
OPEN "CUSTOMER" TO FILE_CUST_FILE ELSE STOP
READ CUST FROM FILE_CUST_FILE, "101" ELSE
  PRINT "Cannot read customer record 101"
  STOP
END
```

The record with key 101 is read into the variable CUST as a dynamic array.

```
CURSOR_NO = ""
SELECT "CUSTOMER" BY "ST" SETTING CURSOR_NO ELSE
  FMSG()
  STOP
END
DONE = FALSE$
LOOP
  READNEXT @ID USING CURSOR_NO BY AT ELSE
    DONE = TRUE$
  END
UNTIL DONE DO
  READ @RECORD FROM CURSOR CURSOR_NO, @ID ELSE
    ERROR_CODE = @FILE.ERROR<1>
    TEXT = ""
    TEXT<-1> = "Record %1% cannot be read!"
    TEXT<-1> = "Error = %2%"
    MSG(TEXT, "", "", @ID:@VM:ERROR_CODE)
    STOP
  END ; * read
  * processing logic here ...
  GOSUB PROCESS
REPEAT
STOP
```

SELECT ... BY is used to initialize a cursor with a sorted list of customer records. A READ-NEXT and READ loop then reads each record in turn, passing the customer records to a local subroutine (not shown) for processing. Note the use of the CURSOR keyword and the cursorvar.

Incorrect Use of READ

```
READ @RECORD ELSE ERROR = @FILE.ERROR<1>
```

No FROM clause is given.

```
READ @RECORD, @ID FROM A ELSE STOP
```

@RECORD, @ID is not a legal variable.

```
READ FROM FILE_CUST, @ID ELSE STOP
```

No variable is given.

READNEXT

The READNEXT statement extracts the next record key from a selected list of record keys (a cursor) until the list is exhausted.

```
READNEXT variable {,value} THEN|ELSE statements
```

This instruction is used to read the keys from a selected list. The list of keys may be from a TCL SELECT command or from an R/BASIC SELECT statement.

PERFORM or SELECT statements may be used in the R/BASIC program to create a list of record keys. For more information, see PERFORM and READ.

variable

If a SELECT or GETLIST is performed at TCL and then used by an R/BASIC program, the next record key in the selected list is accessed each time the READNEXT is executed. The *variable* is assigned the record key from the selected list.

value

When a multivalued sort list is being used, you can use the optional variable *value*. *Value* is used by READNEXT to return the position of *variable* in the multivalued field.

- ☛ For more information about selecting and sorting data, see “Using Filters” in the “Introduction to R/LIST” chapter of *R/LIST*.

THEN

The THEN clause is executed if the READNEXT is successful.

☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if a key cannot be read from the selected list. This is the case, for example, when the list has been exhausted. The STATUS() function indicates the reason for the false branch, and the system variable @FILE.ERROR contains details about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

☞ A READNEXT may return a null record key before the active list is exhausted. Testing for the null key is not a reliable test for list exhaustion.

Correct Use of READNEXT

```
DECLARE SUBROUTINE MSG
OPEN "INV" TO FILE INV ELSE
  MSG("Can't open INV file!","", "", "")
STOP
END
OPEN "DICT INV" TO @DICT
  MSG("Can't open INV dict!","", "", "")
  STOP
END
PERFORM "SELECT INV WITH ST ":QUOTE("WA")
EOF = 0
LOOP
  READNEXT @ID ELSE EOF = 1
UNTIL EOF
  READ @RECORD FROM FILE INV,@ID ELSE
    PRINT "Can't find invoice " :@ID
  END
  PRINT {COMPANY} "L#20" : {INV_NO} "L#15"
REPEAT
PRINT "Done with invoice list"
```

Each sequential key in the selected list of keys from the PERFORM is read into @ID by READNEXT.

Incorrect Use of READNEXT

```
READNEXT ID
The ELSE clause is missing
```

READNEXT BY

To support the use of multiple select lists, an extended READNEXT syntax is provided in addition to the basic version. If the basic syntax is used, the default cursor (cursor 0) is assumed.

READNEXT variable USING cursorvar BY direction ➞

THEN|ELSE statements

variable

Variable is assigned a key from a select list. Each time READNEXT executes, the next available key from the list is assigned to *variable*.

cursorvar

Cursorvar contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement.

- ☛ For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

direction

Direction indicates the logical direction in which the keys are read (ascending or descending) and whether the select list is to be treated as terminating or nonterminating. If the list is treated as non-terminating, the value of STATUS() is set to 1 after the last key has been read.

The value of direction must be an integer between 0 and 3. Alpha codes can be used directly in the READNEXT command for clarity. Possible values:

Value	Code	Meaning
0	AT	Ascending Terminating
1	AN	Ascending Nonterminating
2	DT	Descending Terminating
3	DN	Descending Nonterminating

- ☛ Alphabetic codes are not literal, and so cannot be contained in a variable. The codes can only appear directly in the READNEXT command.

THEN

The *statement(s)* following THEN are executed if a record key is read successfully.

- ☛ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if a key cannot be read from the cursor. This is the case, for example, when the list has been exhausted. The STATUS() function indicates the reason for the false branch, and the system variable

@FILE.ERROR contains detail about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

The ELSE branch is not normally used in nonterminating READNEXT statements, since the list is never exhausted. However, a THEN or ELSE is still required in order for READNEXT to compile properly.

- ✦ Because READNEXT loads @RECORD, @ID and @DICT when resolving a latent list, any program using these variables for its own purposes should save them to local variables before using READNEXT. For more information about latent lists, see "Cursors" in "Programming with R/BASIC" and "Introduction to R/LIST" in *R/LIST*.

Correct use of READNEXT BY

```

DECLARE SUBROUTINE MSG, FMSG
CURSOR.NO = 0
CLEARSELECT CURSOR.NO
SELECT "CUSTOMER" BY "CITY" SETTING CURSOR_NO ELSE
  MSG("Cannot select the CUSTOMER file","","","")
  STOP
END
DONE = 0
LOOP
  READNEXT @ID USING CURSOR.NO BY DN ELSE
    FMSG()
  END
  TEXT="Next key = %1%|Do you wish to continue (Y/N)"
  RESP = "Y"
  MSG( TEXT, "RC", RESP, @ID )
  IF RESP [1,1] EQ "N" THEN DONE = 1
UNTIL DONE REPEAT

```

This program initializes a cursor, then uses READNEXT to move backward in the list. READNEXT tags the cursor as non-terminating, so no ELSE logic is required to find the end of the list (but it is used here to detect serious errors in file access). Instead, a message prompts the user to terminate the program.

```

EQU SPACE$ TO " "
EQU ESC$ TO CHAR(27)
CURSOR NO = 2
CLEARSELECT CURSOR.NO
GOSUB GET_SORT_LIST
SELECT FILE BY SORT_LIST USING CURSOR_NO ELSE
  CALL FMSG("SELECT failure #":@FILE.ERROR<1>)
  STOP
END

DONE = 0 ; RN_MODE = 1
LOOP
  READNEXT @ID USING CURSOR_NO BY RN_MODE ELSE NULL
  READ @RECORD FROM FILE_HANDLE, @ID THEN
    PRINT @ID "L#10" : " " :
    PRINT @RECORD<1> "L#10"

```

```

END
GOSUB GET_INPUT ; * check user's input
BEGIN CASE
  CASE RESPONSE = ESC$
    DONE = 1
  CASE RESPONSE = SPACE$
    IF RN_MODE=1 THEN RN_MODE=3 ELSE RN_MODE=1
END CASE
UNTIL DONE REPEAT

```

This program fragment illustrates a simple browse routine implemented with a bidirectional, nonterminating READNEXT statement. A select list is established using cursor 2. The READNEXT direction is initialized to 1 (ascending nonterminating). User input is checked after each record is displayed. If a user enters a space, direction is toggled to go in the opposite direction. The program terminates when the user presses the [Esc] key.

READO

The READO (“read only”) statement reads a record from a file and assigns its value to a variable.

```

READO variable FROM [filevar
                     [CURSOR cursorvar]] ,key THEN|ELSE statements

```

Using READO

READO is used to read a record that will not be updated. As a consequence, the read request may be fulfilled from any cache or buffers being maintained by the filing system. READO, therefore can be a faster method of accessing data. In a multiuser environment, however, the most current version of a record will not necessarily be returned.

- ☛ The definition of READO (and the use of caches or buffers) is specific to the filing system being used. Many filing systems do not cache data, and therefore implement READ and READO identically.

variable

Variable is assigned the value of the data that is read from the record specified by the *key*.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement and must be preceded with the word CURSOR.

If the file being accessed has had control features added, cursor access will automatically invoke domain conversion during a READO operation.

- ☞ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

filevar

Filevar is a file variable created by a previous OPEN statement.

key

The record referenced by *key* will be read from the (buffered) file identified by *filevar* or the file accessed using the cursor in *cursorvar*.

THEN

The *statement(s)* following THEN are executed if a record is read successfully.

- ☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the record in *variable* cannot be read. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

- ☞ For more information, see OPEN and READ. See also “Dynamic String Arrays” in “Programming with R/BASIC”.

Correct Use of READO

```
OPEN "CUSTOMER" TO CUSTOMER_FILE THEN
  DATA_BYTES = 0
  SELECT CUSTOMER_FILE
  DONE = 0
  LOOP
    READNEXT @ID ELSE DONE = 1
  UNTIL DONE DO
    READO @RECORD FROM CUSTOMER FILE, @ID THEN
      DATA_BYTES += LEN(@RECORD) + LEN(@ID)
    END
  REPEAT
  PRINT "Bytes of data in file = ":DATA_BYTES
END ; * open
```

The program loops through the CUSTOMER file, reading each record. The length of the record and the key are accumulated.

Incorrect Use of READO

```
READO @RECORD FROM CUSTOMER_FILE, @ID
```

No THEN or ELSE clause has been specified.

READV

The READV statement reads data from a specified field of a specified record in an R/BASIC file and assigns its value to a variable. READV is most effective when one field is needed by the program. READ and MATREAD are more efficient when multiple fields are used.

READV variable FROM filevar, key, field THEN|ELSE statements

variable

The designated *variable* is assigned the value of the data from the specified field of the record specified by the key expression.

filevar

The *filevar* is a file variable that has been previously specified in an OPEN statement.

key

Key contains the key to the record that is to be read.

field

The *field* is the index number of the field that is to be read. The value of the *field* expression must be greater than zero.

THEN

The *statements* in the THEN clause will be executed after a successful READV operation.

ELSE

The *statement(s)* following ELSE are executed if the field cannot be read. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

- For more information, see OPEN, READ, MATREAD and WRITE. See also "Dynamic String Arrays" in "Programming with R/BASIC".

Correct Use of READV

```

OPEN "CUSTOMERS" TO FILE.CM ELSE
  CALL MSG("Can't open CUSTOMERS file!", "", "", "", "")
  STOP
END
FOR I = 100 TO 106
  READV CNAME FROM FILE.CM, I, 2 THEN
    CALL MSG(CNAME, "", "", "", "")
  END ELSE
    CALL MSG(I:" not found", "", "", "", "")
  END
NEXT I
END

```

Field 2 of records 100 through 106 is read into CNAME and printed.

Incorrect Use of READV

```

READV CUST FROM Y

```

The ELSE clause is missing.

```

READV FROM TAPE, @ID, 2

```

The variable is missing.

REMOVE

The REMOVE statement is used to extract substrings from dynamic arrays using ASCII characters 249 to 255 as delimiters.

REMOVE variable FROM string AT column SETTING flag

variable, string

The *variable* will contain the substring that is extracted from *string*.

column

The *column* indicates the starting position of the *string* to be extracted. It is important to note that the REMOVE statement changes the column to point to the start of the next substring. The end of the substring occurs when an ASCII character from 249 to 255 is encountered.

flag

The *flag* is set with the following values according to the delimiter found:

Flag	Meaning
0	End of string
1	Record mark ASCII character 255
2	Field mark ASCII character 254
3	Value mark ASCII character 253
4	Subvalue mark ASCII character 252
5	Text mark ASCII character 251
6	ASCII character 250
7	ASCII character 249

- ☞ REMOVE extracts data from a long string faster and more efficiently than EXTRACT, if doing sequential access through the array.

Correct Use of REMOVE

```

STRING = "123" : @VM : "678" : @FM : "ABC"
*
TOP:
PRINT "-----"
PRINT "Enter column number:":
INPUT COL
IF COL = "END" THEN STOP
REMOVE VAR FROM STRING AT COL SETTING FLAG
PRINT VAR
PRINT "FLAG = " :FLAG
BEGIN CASE
  CASE FLAG = 0 ; PRINT "END OF STRING"
  CASE FLAG = 2 ; PRINT "FIELD MARK"
  CASE FLAG = 3 ; PRINT "VALUE MARK"
END CASE
GO TOP

```

This program prompts for a column position and then extracts a substring from between the column and the next delimiter.

Incorrect Use of REMOVE

```

REMOVE X FROM Y SETTING Z

```

The AT clause is missing.

REPLACE

The REPLACE function replaces a field, a value, or a subvalue in a dynamic array.

REPLACE(expression, field, value, subvalue, new)

expression

The *expression* designates the dynamic array that is to be searched.

field, value, subvalue

These are the delimiter expressions. Their respective numeric values determine whether the new data will replace a field, a value, or a subvalue. Note these examples:

```
F = REPLACE (A, 2, 0, 0, NEW)
```

replaces a field. When both the *value* and *subvalue* are 0 (zero), the new data is inserted before the second field (specified by the field) of dynamic array A. The variable F is assigned the new array.

```
V = REPLACE (A, 2, 3, 0, NEW)
```

replaces a value. When only the *subvalue* is 0 (zero), the new data is inserted before the third value of the second field (specified by the value) of dynamic array A. The variable V is assigned the new array.

```
S = REPLACE (A, 3, 2, 1, NEW)
```

replaces a subvalue. When all three delimiter expressions have a non-zero value, the new data is inserted before the first subvalue of the second value of the third field (specified by the subvalue) of dynamic array A. S is assigned the new array.

Field is the highest level delimiter while *subvalue* is the lowest level delimiter. If a higher level delimiter has a 0 (zero) value while a lower level delimiter has a non-zero value, the 0 (zero) delimiter is assumed to be 1 (one). In the following example:

```
REPLACE (A, 0, 0, 2, B)
```

is assumed to be:

```
REPLACE (A, 1, 1, 2, B)
```

If the second, third, or fourth expression has a -1 (minus one) value, the new data is appended after the specified field, value, or subvalue. The expression is not modified by the REPLACE function.

☞ For more information, see < > (angle bracket operators), DELETE, EXTRACT, and INSERT. See also “Dynamic String Arrays” in the chapter “Programming with R/BASIC”.

new

New specifies the new data that is to replace the existing contents.

Correct Use of REPLACE Function

```

* multivalued field with 4 names
NAMES = "ALAN" : @FM : "BRAD" : @FM : "HAL" : @FM : "MIKE"
NAME.TO.CHANGE = ""
CALL MSG("Change which name?", "R", NAME.TO.CHANGE, "")
NEW.NAME = ""
CALL MSG("What is the new name?", "R", NEW.NAME, "")
LOCATE NAME.TO.CHANGE IN NAMES USING @FM SETTING POS THEN
    NEW.NAMES = REPLACE(NAMES, POS, 0, 0, NEW.NAME)
END ELSE
    CALL MSG("No such name!", "", "", "")
END
STOP

```

The program builds an array (field mark delimited) of names. The user is asked to change one name. Based on the user's input, the program searches the name array and replaces the old name with the user's new name.

RETURN/RETURN TO

The RETURN or RETURN TO statement terminates an internal or an external subroutine and returns control back to the main or calling program.

```

RETURN [{expression}]
      [TO label{:}]

```

RETURN

The simple RETURN statement transfers control from an internal or external subroutine back to the statement in the main or calling program that immediately follows the GOSUB or CALL statement.

RETURN expression

The RETURN *expression* form may only be used in functions and formulas in symbolic dictionary items. The result of *expression* is passed back to the calling program as a value that is used by the *expression* from which the call was made.

RETURN TO

The RETURN TO format with the specified label can be used only to exit the internal subroutine initiated by a GOSUB statement. Control of the program will pass directly from the internal subroutine to the specified label within the program. This format should be used very carefully by the programmer; careless use can lead to difficulties in debugging and in future modification of the program.

An error message will result if a non-existent label is specified.

☞ For more information, see CALL, DECLARE, FUNCTION, and SUBROUTINE. See also “External Functions and Subroutines” in “Programming with R/BASIC”.

Using STOP in a subroutine will halt execution of the main program.

Execution of an END statement in an external subroutine has the same effect as a RETURN.

Correct Use of RETURN, RETURN TO Statements

```
SUBROUTINE X(Z)
Y = Z+ 1
GOSUB PRINTIT:
RETURN ; * this returns control to the calling program
```

```
PRINTIT:
PRINT Y,Z
RETURN ; * This returns control
          * to the statement following GOSUB
```

This is an example of a return from a called subroutine and a return from an internal subroutine.

```
* program to get age
BIRTHDAY = ""
CALL MSG("When is your birthday?","R",BIRTHDAY,"")
BIRTHDAY = ICONV(BIRTHDAY,"D2-")
CALL MSG("You are %1% years old!","", "",AGE(BIRTHDAY))
STOP
```

```
FUNCTION AGE(BIRTHDAY)
* returns accurate age based on internal form of birthday
TODAY = OCONV( DATE(), "D2-")
BDAY = OCONV( BIRTHDAY, "D2-")
* calculate years
AGE = TODAY[7,2] - BDAY[7,2]
* has the birthday happened this year?
IF TODAY[1,5] < BDAY[1,5] THEN AGE -= 1
RETURN AGE
```

The main program prompts for a birthday and returns the user's age. The function calculates age by comparing the external form of the birthday and today's date. The calculated age is passed back to the calling program in the RETURN statement.

Incorrect Use of RETURN, RETURN TO Statements

```
RETURN 100
```

The TO is missing. A simple RETURN will be executed.

```
RETURN TO
```

No statement label is used.

RND

The RND function returns a random numeric integer. The random number generator can be seeded using the INITRND statement.

RND (expression)

expression

The RND function returns a random number that is from 0 (zero) to 1 (one) less than the designated number. For instance, in

```
N = RND (501)
```

the variable identifier N will be assigned a random number from 0 to 500. Note that 0 (zero) is a possible number in the random selection.

An *expression* designating null or (0) returns a (0). A negative *expression* returns a negative random number.

Correct Use of RND Function

```
PRINT RND (100)
```

Prints a number between 0 and 99.

```
K = 200
```

```
L = 5
```

```
KXL = RND (K*L)
```

Some number between 0 and 999 will be assigned to the variable identifier KXL.

```
IF X < 100 THEN
```

```
  Y = RND (X)
```

```
END
```

If the value of X is less than 100, then the value of Y will be between 0 and (X-1).

Incorrect Use of RND Function

```
A = RND (1001)
```

Uses a space between the function and parentheses.

```
K = RND (5+"XYZ")
```

An attempt has been made to use alphabetic characters in an arithmetic expression.

```
RND (501) = S
```

The function appears on the left side of the assignment operator.

SELECT

The **SELECT** statement creates a list of record keys from a given file. The record keys are used by a subsequent **READNEXT** statement.

```
SELECT filevar
```

SELECT

SELECT is very similar to an **R/LIST SELECT** sentence. **R/BASIC SELECT** is faster than **R/LIST SELECT** because **R/LIST** must read through the entire file before record keys are available. However, selection and sort criteria cannot be specified. For more information on accessing Advanced Revelation files, see the topic “Advanced Revelation File Access” in the chapter “Programming with **R/BASIC**”.

An extended form of **SELECT** is available for specifying sort and selection criteria. For more information, see **SELECT BY** in this section.

filevar

The *filevar* refers to a file variable name that has been specified in a previous **OPEN** statement.

- ☞ Do not add records to a file when using **SELECT**. Results are unpredictable. Use the **R/LIST SELECT** with **PERFORM** when adding new records.

For more information, see **PERFORM**. See also **SELECT** in “The Command Level” section.

Correct Use of SELECT Statement

```
DONE = 0
OPEN "VOC" TO FILE.VAR THEN
  SELECT FILE.VAR
  LOOP
    READNEXT ID THEN
      PRINT ID
    END ELSE DONE = 1
  UNTIL DONE REPEAT
END
STOP
```

The program prints the keys of the VOC file.

Incorrect Use of SELECT Statement

```
SELECT "VOC"
"VOC" is not a valid filevar. SELECT is not used with literal file names.
```


SELECT BY

To support the use of multiple select lists and sort options, an extended version of the SELECT statement is available. If the basic syntax is used, R/BASIC will use the default cursor.

- For more information on cursors, see “Cursors” in the chapter “Programming with R/BASIC”.

```
SELECT file_expression BY sort_list cursor_keyword cursorvar
    THEN|ELSE statements
```

file_expression

File_expression designates the file that is to be accessed by the SELECT statement. Unlike the basic SELECT syntax, this file name is provided as a literal file name, not as a file variable generated by an OPEN statement.

sort_list

The *sort_list* is used to specify the sort order for the record keys to be returned in the cursor.

- The *sort_list* is used to trigger the extended syntax for the SELECT statement. You must use BY to make use of the extended syntax, even if you do not wish to sort the list of record keys. If you do not wish to specify a sort order for keys in the cursor, but still wish to use the extended syntax for SELECT, specify a null ("") for *sort_list*.

The *sort_list* consists of a field mark delimited array of field names. The field names must resolve to the names of fields in the file represented by *file_expression*. The record keys in the cursor will be sorted according to the field(s) in *sort_list*.

The default sort order for any field specified in the *sort_list* is an ascending sort. To indicate a descending sort for any field, precede the field name with # (pound sign). For example, to indicate reverse sort order for the field COMPANY, use the expression #COMPANY.

Sort priority is determined by the order of the elements in the *sort_list*. The first element in the list will be the primary sort criterion, the second element the secondary criterion, etc.

cursor_keyword

A *cursor_keyword* determines the disposition of the cursor after SELECT has sorted the key list.

There are four possible keywords:

SETTING
USING
ASSIGNING
MODE *variable, cursorvar*

SETTING *Setting* will find the next available cursor, establish the select list in that cursor, and return the cursor number in that cursor. SETTING will return a false branch (ELSE) if no more cursors are available.

USING *Using* allows re-sorting an active list by specifying new sort criteria in the *sort_list*. The cursor specified in USING must have been assigned the same file as the file in the *file_expression*. If the cursor has been set up for a file and then a SELECT ... BY is performed on that cursor using a different file, the cursor will be cleared and reset. All previous sort and selection criteria are lost.

ASSIGNING *Assigning* allows the key list in a cursor to be assigned to a different file name. ASSIGNING thus functions like USING, but permits the list in a cursor to be re-sorted and applied to a different file.

MODE *Mode* differs from the other keywords in that it permits the designation of a cursor keyword (SETTING, USING, or ASSIGNING) with a variable. This allows a program to determine the proper keyword for SELECT at program execution time.

The MODE syntax is:

MODE *variable, cursorvar*

The value of the variable used must be an integer designating the proper keyword. The values to use for each keyword are:

Value	Keyword
1	SETTING
2	USING
3	ASSIGNING

cursorvar

The *cursorvar* identifies the cursor to use. The SELECT command initializes this variable with information used for subsequent access to the cursor.

THEN

The *statement(s)* following THEN are executed if the cursor is properly initialized.

For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the SELECT operation is not successful. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

Correct use of extended SELECT

```
DECLARE SUBROUTINE MSG, FMSG
SORT LIST = "ZIP" : @FM : "COMPANY"
SELECT "CUSTOMER" BY SORT_LIST SETTING CURSOR ELSE
  FMSG()
  STOP
END
```

This program fragment creates a select list out of records in the CUSTOMER file. The list is sorted according to the fields ZIP and COMPANY. The resultant select list will be assigned to the next available cursor. The cursor number for the cursor assigned by SETTING is passed back to the program in CURSOR. If no cursors are available, or if the sort is not successful, the program displays a message and stops.

```
PERFORM "SELECT CUSTOMER WITH ST = 'CA'"
PERFORM "SAVELIST SAMPLE"
DONE = 0 ; CURSOR_NO = 0
LOOP
  SORT LIST = ""
  GOSUB GET_SORT_FIELDS
  IF SORT_LIST = "" THEN DONE = 1
UNTIL DONE
  PERFORM "GETLIST SAMPLE"
  SELECT "CUSTOMER" BY SORT_LIST USING CURSOR.NO ELSE
    FMSG()
    STOP
  END
  GOSUB DISPLAY
REPEAT
CLEARSELECT CURSOR_NO
```

In this program fragment, the same list of customers is sorted repeatedly using a SELECT... BY statement. The list is initially created using a PERFORM SELECT statement, and saved using SAVELIST. Thereafter, the user is prompted to enter the list of fields by which to sort the data (prompting actually occurs in a subroutine not listed here). The program activates

the list using a TCL-level GETLIST statement, and then sorts the list using a SELECT ... BY statement. The sorted list is passed to a processing subroutine. The program terminates when the sort list is returned blank.

```

OPEN "BENEFITS" TO BENEFITS_FILE ELSE
  CALL MSG("No BENEFITS file!", "", "", "")
  STOP
END
* select employees by salary (highest to lowest)
SELECT "EMPLOYEE.MASTER" BY "#SALARY" USING 1 ELSE
  CALL MSG("SELECT failure #":STATUS(), "", "", "")
END
* assign that list to a new file
SELECT "BENEFITS" BY "" ASSIGNING 1 ELSE
  CALL MSG("SELECT failure #":STATUS(), "", "", "")
END
*
DONE = 0
LOOP
  READNEXT @ID USING 1 BY AT ELSE DONE = 1
UNTIL DONE DO
  GOSUB PROCESS_RECORD
REPEAT

```

This program fragment illustrates the use of the ASSIGNING keyword. In this example, the first SELECT establishes a select list (in descending order by salary) of records in the EMPLOYEE.MASTER file. The second SELECT statement assigns the BENEFITS file to this cursor. This permits the processing loop to use the keys from the EMPLOYEE.MASTER file to read records from the BENEFITS file. In this example, it is assumed that the files share a common key structure.

SEQ

The SEQ function returns the numeric value of an ASCII character.

SEQ(expression)

expression

The first character of the value that is specified in the *expression* is converted from its ASCII character code to its corresponding numeric value. Any constant, variable identifier, or another intrinsic function may be specified by the expression. For example:

```
NUM = SEQ("B")
```

assigns the decimal equivalent of the string value "B" to the variable identifier NUM. The chart listing the ASCII codes in the *Quick Reference* manual shows that the ASCII character code B has a decimal code value of 66. Therefore, NUM would be assigned the value 66.

In the following, more complex example, the decimal equivalent of each individual character of the character string REVELATION is encoded in the matrix V elements.

```
DIM V(10)
N = "REVELATION"
FOR I = 1 TO LEN(N)
    V(I) = SEQ(N[I,1])
NEXT I
```

For more information, see CHAR.

Correct Use of SEQ Function

```
PRINT SEQ("1")
Prints the number 49.
```

```
COMP = SEQ(F)
The value of the variable identifier F is converted from its ASCII character code to its numeric value and assigned to COMP.
```

```
PRINT "Press [Esc] to exit."
LOOP
    INPUT X,-1
UNTIL X EQ CHAR(27) DO
    PRINT "The ASCII value of this key is:"
    PRINT "Character 1 = " :
    PRINT SEQ(X[1,1]) ; * ASCII value of chr 1
    PRINT "Character 2 = " :
    PRINT SEQ(X[2,1]) ; * ASCII value of chr 2
REPEAT
```

The character or characters entered with each keystroke have their ASCII values printed. The program is stopped by entering [Esc].

Incorrect Use of SEQ Function

```
SEQ(Q) = T
The function appears on the left side of the assignment operator.
```

SERIAL()

The SERIAL() function returns the Advanced Revelation serial number.

SERIAL ()

SERIAL()

The SERIAL() function returns the string value of the current serial number. The SERIAL() function can be used to limit the use of application software written in R/BASIC to a specific serial numbered copy of Advanced Revelation.

The serial number may be read but cannot be modified.

Correct Use of SERIAL() Function

```
IF SERIAL() NE 299 THEN STOP
```

Program will stop if serial number 299 is not returned.

```
PRINT SERIAL()
```

This will print the Advanced Revelation serial number for the copy being used.

Incorrect Use of SERIAL() Function

```
SERIAL() = 1048
```

The function appears on the left side of the assignment operator.

```
PRINT SERIAL(1)
```

An argument has been used where none is allowed.

SIN

The SIN function calculates the trigonometric sine of an angle.

SIN(expression)

expression

The *expression* must be a numeric value that designates degrees. The values must be within the range 0 to 360.

The following is an example of a program using SIN and its resulting output:

```
PRINT "ANGLE", "SIN(ANGLE) "  
FOR ANGLE = 0 TO 360 STEP 45  
  PRINT ANGLE, SIN(ANGLE)  
NEXT ANGLE  
END
```

ANGLE	SIN(ANGLE)
0	0
45	.7071
90	1
135	-.7071
180	-0
225	-.7071
270	-1
315	-.7071
360	0

Correct Use of SIN Function

`TAN = SIN(X) / COS(X)`

The sine of X is divided by the cosine of X and assigned to the variable identifier TAN.

Incorrect Use of SIN Function

`A = SIN X`

Parentheses are missing.

SPACE

The SPACE function generates a string value containing a specified number of blank spaces.

SPACE (expression)

expression

The *expression* must be a numeric value. Up to 65,530 blank spaces may be specified. If the *expression* evaluates to less than 0 (zero), the results are the same as 0.

In this example:

```
PRINT "CUST.NO DESC":SPACE(16) : "INVOICE.NO"
```

the printout would be:

```
CUST.NO DESC          INVOICE.NO
```

Sixteen blank spaces would follow the character string value enclosed in quotes.

For more information, see STR.

Correct Use of SPACE Function

```
D = 12
F = SPACE(D)
```

The variable identifier F is given a string value containing 12 blank spaces.

```
S = SPACE(4)
L = "DOE"
M = "E."
F = "JANE"
N = S:F:S:M:S:L
```

Assigns the concatenated string to the variable identifier N. This would read "^^^JANE^^^E.^^^DOE" (The ^ represents blank spaces)

```
DIM TABLE(12)
MAT TABLE = SPACE(10)
```

Each of the elements of the array TABLE is assigned a string value containing 10 blank spaces.

Incorrect Use of SPACE Function

```
X = SPACE *2
```

The parentheses and argument are missing.

```
TAB = SPACE
```

The expression is missing.

SQRT

The SQRT function calculates the square root of a positive number.

SQRT (expression)

expression

The *expression* must evaluate to a positive number or 0 (zero). A warning message will appear if the *expression* evaluates to a negative value.

The following example is a simple program to calculate square roots:

```
PRINT "I", "SQRT(I)"
FOR I = 0 TO 30 STEP 3
    PRINT I, SQRT(I)
NEXT I
END
```


I	SQRT(I)
0	0
3	1.732
6	2.4495
9	3
12	3.4641
15	3.873
18	4.2426
21	4.5825
24	4.899
27	5.1962
30	5.4772

Correct Use of SQRT Function

C = SQRT(144)
C is set to a value of 12.

Incorrect use of SQRT Function

PRINT SQRT[144]
Brackets are used where parentheses are required.

Y = SQRT 10
The parentheses are missing.

STATUS()

The STATUS() function tests the results of a previous operation to determine its validity.

STATUS()

STATUS()

The STATUS() function returns a value that indicates the status of the previous operation. No argument is defined within the parentheses.

Certain operations set and use the STATUS value. However, you can assign a STATUS value for your own purposes. For example:

```
STATUS() = 1
```

will assign the value 1. Note that this value will be changed if a call to one of the operations that set STATUS precedes your reference. For more information, see READ, READO, READV, MATREAD, WRITE, WRITEV, MATWRITE, CLEARFILE, LOCK, UNLOCK, OPEN, DELETE, SELECT, READNEXT, ICONV, OSBREAD, OSBWRITE, OSCLOSE, OSDELETE, OSOPEN, OSREAD, OSWRITE, and SUSPEND.

Correct Use of STATUS() Function

```
PROMPT ""
ERROR = ""
ERROR<1> = "Successful open - no error"
ERROR<2> = FILE : " is a bad filename"
ERROR<3> = "Access to " : FILE : " denied by OS"
ERROR<5> = FILE : "is not an OS filename"
ERROR<6> = "Undefined error"
```

```
LOOP
  PRINT "Enter operating system filename:":
  INPUT FILE
UNTIL FILE EQ "END"
  OSOPEN FILE TO FILE.VAR THEN
    OSBREAD FRAME FROM FILE.VAR AT 0 LENGTH 25
  OSCLOSE FILE
END ELSE
  PRINT ERROR<STATUS()+1>
END
REPEAT
```

The error messages are assigned to the dynamic array ERROR. STATUS() is set when each operating system file is read. The number returned by STATUS() is used to extract the message to be printed.

Incorrect Use of STATUS() Function

```
X = STATUS
```

Parentheses are missing.

STOP

The STOP statement terminates program execution. The statement may appear anywhere in the program.

STOP {expression}

expression

The *expression* will be printed on the terminal screen by the system before the STOP is executed.

When STOP is executed in a subroutine, the main program terminates.

Correct Use of STOP Statement

```
OPEN "CST" TO CUST.FILE ELSE
  STOP "Can't Open Customers"
END
PRINT "Input KEY" : ; INPUT KEY
```

```

READ CUST.REC FROM CUST.FILE , KEY ELSE
  STOP "Can't read " :KEY: " from Customers File"
END
STOP

```

STOP is used here to indicate unsuccessful OPEN and READ statements.

Incorrect Use of STOP Statement

```

STOP GOSUB 103

```

Only an expression can be used with STOP.

STR

The STR function returns a string data value that contains a specified string expression repeated a specified number of times.

STR(expression1, expression2)

expression1

Expression1 specifies the string value that is repeated. The expression must be a variable name or character string value.

expression2

Expression2 must be a positive integer or must evaluate to a positive integer. *Expression2* specifies the number of times to repeat *Expression1*. In the following example, the variable identifier X is assigned the string value containing 10 asterisk characters:

```
X = STR("*",10)
```

The first and second *expressions* are separated only by a comma with no spaces permitted in the format. In the following example, the variable identifier Y would contain "101101101":

```
Y = STR("101",3)
```

See also SPACE.

Correct Use of STR Function

```
X = STR("_",60)
```

The variable identifier X is given the value of 60 underlines.

```
E = "XX"
```

```
G = STR(E,5)
```

G is assigned the string value of XXXXXXXXXX.

```

EQU CRN.UL TO CHAR(201)
EQU CRN.LL TO CHAR(200)
EQU CRN.UR TO CHAR(187)
EQU CRN.LR TO CHAR(188)
EQU LN.H TO CHAR(205) ; * Horizontal
EQU LN.V TO CHAR(186) ; * Vertical
EQU CRLF TO CHAR(13) : CHAR(10)
EQU FILL.SP TO SPACE(18)
TOP.LN = CRN.UL : STR(LN.H,18) : CRN.UR : CRLF
BTM.LN = CRN.LL : STR(LN.H,18) : CRN.LR
LOOP
  PRINT @(0,23) : "INPUT START COL (0 - 50) " : ; INPUT COL
UNTIL COL = "END"
  PRINT @(0,23) : @(-4) : ; * CLEAR INPUT LINE
  PRINT "INPUT STARTING ROW (0 - 13) " : ; INPUT ROW
  PRINT @(0,23) : @(-4) : ; * CLEAR INPUT LINE
  MIDDLE.SP = SPACE(COL)
  MIDDLE = STR(MIDDLE.SP : LN.V : FILL.SP : LN.V : CRLF,10)
  BOX = @(COL,ROW) : TOP.LN : MIDDLE : @(COL,ROW+11) : BTM.LN
  PRINT BOX
  PRINT @(0,23) : "PRESS [Enter] TO CONTINUE " : ; INPUT X
  PRINT @(-1) ; * CLEAR SCREEN
REPEAT
END

```

The program draws a box and fills it.

Incorrect Use of STR Function

```
N = STR(NAME, 3)
```

If NAME is a string value, it must be enclosed in quote marks.

SUBROUTINE

A SUBROUTINE statement identifies a program as an external subroutine. It must be the first line in the subroutine and is used to specify the name and arguments that other programs must use when entering or calling the subroutine.

```
SUBROUTINE name { (argument [{,argument} ::]) }
```

SUBROUTINE

The subroutine must have been previously compiled and cataloged before it can be called. Use the TCL command MAKEVOC to catalog a subroutine. Use RETURN to return control to the calling program.

Once a subroutine has been called, it remains in memory until the main program that called it executes a STOP statement, or until a RESET or debugger END com-

mand is executed. For information on removing subroutines from memory, see EXPENDABLE.

name

Any legal variable identifier may be used as the *name* of the subroutine.

argument

An argument may be a variable, matrix, constant, or expression that is used to pass values to and from the calling program and the subroutine. The calling program must use the same number of arguments or fewer arguments than specified in the subroutine to be called. If used, the *arguments* must be enclosed in parentheses.

The *arguments* may consist of zero or more expressions, separated by commas. If no arguments are used, the parentheses are omitted.

For more information, see CALL, COMMON, DECLARE, EXPENDABLE, FUNCTION, and RETURN. See also "Subroutines" in the chapter "Programming with R/BASIC". See also TCL and MAKEVOC in "The Command Level".

Correct Use of SUBROUTINE Statement

```

DECLARE SUBROUTINE ERROR
EQU TRUE$ TO 1
EQU FALSE$ TO 0
DONE = FALSE$
OPEN "CUSTOMER" TO FILE_CUST ELSE
  TXT = "File not attached"
  ERROR("MAIN", "CUSTOMERS", "", TXT)
  DONE = TRUE$
END
IF DONE ELSE
  SELECT FILE_CUST
  LOOP
    READNEXT KEY ELSE
      DONE = TRUE$
    END
  UNTIL DONE
  READ REC FROM FILE_CUST, KEY ELSE
    ERROR("MAIN", "CUSTOMERS", KEY, "Record not found")
  END
  REPEAT
END
STOP

SUBROUTINE ERROR(PROG_NAME, FILE_NAME, KEY, TEXT)
OPEN "ERROR_LOG" TO FILE_LOG THEN
  LOG_REC = ""
  LOG_REC<1> = PROG_NAME
  LOG_REC<2> = FILE_NAME
  LOG_REC<3> = KEY
  LOG_REC<4> = @USERNAME
  LOG_REC<5> = TEXT

```

```

LOG_KEY<6> = DATE() : "*" : TIME() : "*" : @STATION
WRITE LOG_REC ON FILE_LOG, LOG_KEY ELSE
    TEXT = "No error entry"
END
END ELSE
    TEXT = "No error log"
END
RETURN

```

This routine is an error logger. It will log the event of an error for any program calling it. The key to the error file is the date and time of the error and the station number of the workstation experiencing the error. The error record can contain (depending on what is passed to the routine) the program name, the file name, the key to the file, the user name, and any text the programmer may want to have recorded. Note that because the calling program may not have any control over whether the error log file is attached, the routine sets TEXT equal to the string "No error log" if the error log file cannot be opened. Programs that need to know whether or not an error was logged should check the value of TEXT.

Incorrect Use of SUBROUTINE Statement

```

TT = C+B
SUBROUTINE ACTT(H)

```

The SUBROUTINE statement is not on the first line.

```

CALL TEST(1,2,3)
SUBROUTINE TEST(X,Y)

```

The number of arguments is not the same in CALL and SUBROUTINE.

SUM

The SUM function adds the numeric data at the lowest delimiter level, of a dynamic array to the next higher level of the array.

SUM(expression)

expression

The *expression* designates the dynamic array in which the numeric data is to be summed. All elements must contain numeric data. The computer searches the specified array to find its lowest level delimiter. The numeric data of that lowest level is then added to the numeric value of the next higher level, and that sum is stored there.

In the following example:

```
A = "2n1222"
SUM(A)
```

subvalue 1 (one) is the lowest level, and its value would be added to and stored in the next value. The result would be 3²2².

Each single SUM function manipulates data only to one level higher in the existing string each time. Therefore, the SUM function needs to be executed repeatedly to raise a multi-level array to a single numeric value.

SUM recognizes seven levels of delimiters: ASCII character 248 through ASCII character 254.

Correct Use of SUM Function

```
X = 5 : @FM : 4 : @VM : 3 : @FM : 6
PRINT SUM(X)
```

Prints 5^7^6, where "^" represents a field mark (ASCII 254).

```
PRINT SUM(SUM(X))
```

Prints 18.

Incorrect Use of SUM Function

```
X = 5 : @FM : "text" : @VM : 4 : @FM : 7
Z = SUM(X)
```

The dynamic array contains a non-numeric element.

```
SUM(2) = X
```

The function appears on the left side of the assignment operator.

```
Q = SUM(X, Y)
```

Uses two expressions where only one is permitted.

SUSPEND

The SUSPEND statement allows the R/BASIC program to execute operating system commands that require more memory than is available while Advanced Revelation is running.

SUSPEND expression

SUSPEND

Advanced Revelation can write its present state to disk, release memory for the next operation, and call the operating system.

The SUSPEND statement allows a user to perform any operating system command in an R/BASIC program. The *expression* must yield the operating system command that is to be performed.

SUSPEND writes the current state of Advanced Revelation to an operating system file before executing the specified operating system command. Advanced Revelation is read back into memory when the command is done. PCPERFORM leaves Advanced Revelation in memory and is faster.

☞ PCPERFORM and SUSPEND function identically in OS/2 versions.

expression

The *expression* is executed as though it had been typed at the operating system default drive prompt. The operating system command processor must be accessible to the operating system for SUSPEND to work. Under MS-DOS and OS/2, the environmental variable COMSPEC must be set.

STATUS()

STATUS() may be used to verify the success of the SUSPEND statement.

STATUS() returns the following codes:

Value	Meaning
0	SUSPEND statement was successful
-1	Invalid function code
-2	Invalid file specifications
-3	Invalid path specifications
-29	Write fault error or full disk
-80	Rollout file already exists

When Advanced Revelation is re-entered after performing operating system commands, the path is reset to its previous status before the execution of the SUSPEND.

☞ The SUSPEND statement reads the operating system filename from the system variable @ROLLOUT.FILE. Set @ROLLOUT.FILE using the command ROLLOUT before using SUSPEND.

Warning! Do not execute any programs that will remain resident in memory upon termination. These may interfere with Advanced Revelation.

For more information, see PCPERFORM. See also PC, ROLLOUT, and SUSPEND in *TCL*. See your operating system manual for command information.

Correct use of SUSPEND Statement

```
CMD = "DIR"
LOOP
  SUSPEND CMD
  PRINT "ENTER OS COMMAND":
  INPUT CMD
UNTIL CMD = "END" REPEAT
```

The program will perform operating system commands until END is typed. Advanced Revelation will be written to an operating system file before each operating system command is performed. The Advanced Revelation environment will be read back in memory when the operating system commands have executed.

Incorrect Use of SUSPEND Statement

```
SUSPEND
```

SUSPEND requires an operating system command.

SWAP

The SWAP statement is used to perform global replacements of one string with another. The SWAP statement allows for replacement of strings of data; the CONVERT statement replaces data character for character.

For more information, see CONVERT.

SWAP expression1 WITH expression2 IN variable

SWAP

SWAP will replace all occurrences of *expression1* with *expression2* in the specified *variable*.

Correct Use of SWAP Statement

```
X = "ABCDEF"
SWAP "BC" WITH "XYZ" IN X
The string ABCDEF will become XYZDEF.
```

```
SWAP "BCA" WITH "" IN "ABCABCABC"
The string ABCABCABC will become ABC.
```

```
X = "AAAAA"
SWAP "AA" WITH "B" IN X
The string AAAAA will become BBA.
```

```

X = "THIS IS MONDAY MORNING"
LOOP
  PRINT X
  PRINT "Input word to change: " :
  INPUT CHANGE
UNTIL CHANGE = "END"
  PRINT "Input new word for ":QUOTE(CHANGE)
  INPUT REPLACE
  SWAP CHANGE WITH REPLACE IN X
REPEAT
STOP

```

The program prompts the user to change any character(s) in the string "THIS IS MONDAY MORNING."

Incorrect Use of SWAP Statement

```

SWAP BCA IN ABCABC

```

No with expression is specified.

TAN

The TAN function produces the tangent of an angle expressed in degrees.

TAN(expression)

expression

The *expression* must be numeric and must express degrees.

Using the TAN function in the following example:

```

PRINT "ANGLE", "TAN(ANGLE) "
FOR ANGLE = 0 TO 360 STEP 45
  IF ANGLE = 90 OR ANGLE = 270 THEN
    PRINT ANGLE, "Undefined"
  ELSE
    PRINT ANGLE, TAN(ANGLE)
  END
END
NEXT ANGLE

```

produces the following results:

ANGLE	TAN(ANGLE)
0	0
45	1
90	Undefined
135	-1
180	0

225	1
270	Undefined
315	-1
360	0

Correct Use of TAN Function

T = TAN (DEGREES)

T is set to the tangent of the value of variable identifier DEGREES.

Incorrect Use of TAN Function

TANGENT = TAN (A, B)

The extra expression used is illegal.

TAN (X) = 5

The function appears on the left side of the assignment operator. "TAN" has not been dimensioned or declared as a function.

TIME()

The TIME() function returns the internal time of day.

TIME ()

TIME

The TIME() function returns the string value of the internal time of day. Internal time is calculated as the number of seconds past midnight. For example,

T = TIME ()

assigns the string value of the internal time to the variable identifier T.

For example, if TIME() were used at 10:30:12 (approximately 10:30 AM), it would return the internal time of 37812 seconds after midnight.

The following sample shows the number of seconds for common times:

External	Internal
Midnight	0
6:00 AM	21600
Noon	43200
One hour	3600

To convert the internal time value to hour and minute value, refer to OCONV Time (MT), ICONV Time (MT), and TIMEDATE().

Correct Use of TIME() Function

```
CLOCK = TIME ()  
The string value of internal time is assigned to CLOCK.  
  
PRINT OCONV (TIME () , "MT")  
Prints the time in external format.  
  
IF TIME () > 43200 THEN  
  GOTO 50  
END  
Program branches to statement label 50 if more than 43200 seconds have passed since mid-  
night.
```

Incorrect Use of TIME() Function

```
T = TIME (4)  
Uses an expression where none is permitted.  
  
TIME ()  
TIME is used alone. (A function cannot be used alone as a statement.)
```

TIMEDATE()

The TIMEDATE() function returns a string value expressing the current time and date in external format.

TIMEDATE ()

TIMEDATE()

The string value containing the current time and the current date is returned in the following format:

HH:MM:SS DD MMM YYYY

Value	Meaning
HH	Hours in 24-hour digital time
MM	Minutes
SS	Seconds
DD	Day
MMM	Month, abbreviated to three letters
YYYY	Year

The time is expressed in 24-hour days, with colons separating the hours, minutes, and seconds. The designations AM and PM are not used.

The date is expressed as the annual date, with spaces separating the day, month, and year. In the following example:

```
TD = TIMEDATE()
```

the time and date are assigned to the variable identifier TD. If the function is executed at 1:45 PM on December 1, 1982, the string value 13:45:00 1 Dec 1982 will be assigned.

✖ An argument expression is illegal when using the TIMEDATE() function.

For more information, also see DATE() and TIME().

Correct Use of TIMEDATE() Function

```
PRINT TIMEDATE()
```

Prints the current time and date.

```
NOW = TIMEDATE()
```

Assigns the string value of time and date to the variable NOW.

Incorrect Use of TIMEDATE() Function

```
X = TIMEDATE
```

The parentheses are missing.

```
Y = TIME DATE()
```

TIME and DATE are used as two words.

TRANSFER

TRANSFER moves the contents of one variable to another and clears the original variable.

```
TRANSFER variable1 TO variable2
```

TRANSFER

The TRANSFER statement moves the value of a variable by reference. Instead of copying the data represented by a variable, TRANSFER copies the pointer to that data. The effect is a very quick reassignment, with no loss of memory.

If the contents of a variable are copied using an assignment operator (for example, A=B), R/BASIC makes a copy of the value in the source variable (B), and loads it into the location represented by the target variable (A). The assignment operator is a transfer by value. This consumes both time and memory and is thus less efficient than using the TRANSFER command.

variable1

Variable1 is the source variable. After the execution of a TRANSFER statement, *variable1* is null.

variable2

Variable2 is the target. After a transfer, *variable2* contains the data originally represented by *variable1*. It is not necessary to initialize the target variable before transferring to it.

Correct Use of TRANSFER Statement

```

DECLARE SUBROUTINE MSG
OPEN "TEST" TO CUSTOMER_FILE ELSE
  FMSG()
  STOP
END
SELECT CUSTOMER_FILE
DONE = 0
LOOP
  READNEXT @ID ELSE DONE = 1
UNTIL DONE
  READ @RECORD FROM CUSTOMER_FILE, @ID THEN
    * save contents before calling routine
    * in case RECORD_STATS changes values
    TRANSFER @RECORD TO SAVE_RECORD
    TRANSFER @ID TO SAVE_ID
    CALL RECORD_STATS
    * restore contents
    TRANSFER SAVE_RECORD TO @RECORD
    TRANSFER SAVE_ID TO @ID
    CALL MSG(@RECORD:@FM:@ID,"","","")
    GOSUB FURTHER_PROCESSING
  END
REPEAT
STOP

```

The program processes each record in the CUSTOMER file. Before calling the external program RECORD_STATS, the program saves the values of @RECORD and @ID by transferring them to temporary variables. The values are restored after the call. Note: the local subroutine FURTHER.PROCESSING is not shown.

Incorrect Use of TRANSFER Statement

```
TRANSFER "string" TO B
```

TRANSFER can only use variables as its arguments, not expressions.

TRIM, TRIMF, TRIMB

The TRIM function deletes the extra blank spaces from a character string value or variable.

```
TRIM(expression)
TRIMF(expression)
TRIMB(expression)
```

TRIM

Multiple consecutive spaces can be deleted from a character string or variable by using the TRIM function. The unneeded blanks preceding the first valid character string and those extra blanks following the last valid characters are deleted along with any extra blanks separating valid words.

In the following examples, ^ indicates a blank space:

```
T = "^^^^^EXTRA^^^BLANKS^^TAKE^^^UP^^SPACE^^^"
T = TRIM(T)
PRINT T
```

The extra blanks will be trimmed and the variable identifier T will be printed as:

```
EXTRA^BLANKS^TAKE^UP^SPACE
```

TRIMF

The TRIMF function deletes spaces from the front of a character string.

Using the same example as above, PRINT TRIMF(T) would print:

```
EXTRA^^^BLANKS^^TAKE^^^UP^^SPACE^^^
```

TRIMB

The TRIMB function deletes blank spaces at the end of a character string.

Using the same example as above, PRINT TRIMB(T) would print:

```
^^^^EXTRA^^^BLANKS^^TAKE^^^UP^^SPACE
```

Correct Use of TRIM Functions

```
SA = "Dear^Mr.^Leu^^^:"
SA = TRIM(SA)
PRINT SA
SA is printed as "Dear Mr. Leu:"
```

```

F = "^^ROBERT^^"
M = "E."
L = "LEWIS.^^^"
NAME = TRIM(F:M:L)
The name is trimmed to the string value "ROBERT^E.^LEWIS."

X = "^^^1^^^2^^3^4^^"
PRINT TRIMF(X)
Prints1^^^2^^3^4^^

PRINT TRIMB(X)
Prints^^^1^^^2^^3^4

```

Incorrect Use of TRIM Functions

```

PRINT TRIM
The expression is not specified.

XYZ = TRIM(X, Y, Z)
Uses three expressions where only one is permitted.

```

UNLOCK

The UNLOCK statement releases a lock set by the station in a previous LOCK statement. After the UNLOCK command has been issued, other users in the network may then lock the file or record.

UNLOCK has no effect unless used on a local area network. Therefore, UNLOCK can be built into applications whether they will be used on networked Advanced Revelation systems or not. Applications not running on a network will simply ignore the UNLOCK commands.

Termination of the program does not unlock any locks that have been set. Use UNLOCK to release each lock. Logging off the system will release all locks. In addition, all locks are cleared when a user resets the system using the TCL RESET command or the END command at the debugger prompt.

UNLOCK filevar ALL CURSOR cursorvar {,key {,locktype}} THEN ELSE statements

filevar

The *filevar* refers to a file variable that has been locked in a previous LOCK statement.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement.

- ✎ For more information about cursors, see “Cursors” in the chapter “Programming with R/BASIC”. See also SELECT ... BY.

key

If a record is to be unlocked, the *key* specifies which record within a file is to be unlocked. To unlock a file locked previously, pass a null as the key expression.

To UNLOCK all LOCKs previously set by this workstation during the current logon session, use the UNLOCK ALL statement.

locktype

The *locktype* specifies the type of unlocking that is to take place. *Locktype* can be passed as a mnemonic code (not a literal) or as a two-digit code. For details on the possible values for *locktype*, see the LOCK command. Note that the code *n0* is used to indicate an UNLOCK ALL.

THEN

The *statement(s)* following THEN are executed if the file or record is unlocked successfully.

- ✎ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the file or record cannot be unlocked. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains details about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

- ✎ For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in an UNLOCK statement are optional. However, it is strongly recommended that all UNLOCK logic be constructed to accommodate possible unsuccessful unlocks. If no THEN or ELSE clause is specified, any errors during the UNLOCK operation are passed to the system error handling routine.

For more information, see LOCK.

Correct Use of the UNLOCK Statement

```
EQU TRUE$ TO 1
EQU FALSE$ TO 0
DECLARE SUBROUTINE MSG
OPEN "CUSTOMER" TO FILE_CUST ELSE STOP
```

```

KEY = ""
MSG("Input record key to read","R",KEY,"")
LOCKED = FALSE$
LOOP UNTIL LOCKED
  LOCK FILE_CUST, KEY THEN
    LOCKED = TRUE$
  END ELSE
    MSG("Record ":KEY:" is already locked!","", "", "")
  END
REPEAT
  READ CUST_REC FROM FILE_CUST, KEY THEN
    MSG("Company name is ":CUST_REC<1>","", "", "")
  END
UNLOCK FILE_CUST, KEY ELSE
  FMSG()
END
STOP

```

Incorrect Use of the UNLOCK Statement

```
UNLOCK "CUSTOMER", 101
```

A file variable must be used, not a file name.

WRITE

The WRITE statement creates or updates a record on disk. If the record does not exist, a new record is created in the file. If a record does exist, it is overwritten.

```

WRITE expression [ON] [filevar] ,key ⇨
                  [TO] [CURSOR cursorvar]

THEN|ELSE statements

```

expression

The *expression* evaluates to the record that is to be written to the file.

filevar

The *filevar* refers to a file variable that has been opened in a previous OPEN statement.

cursorvar

If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement, and must be preceded with the word CURSOR.

If the file being accessed has had control features added, cursor access will automatically use domain validation and conversion during a WRITE operation.

- ☞ For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

key

The *key* specifies the key of the record to be written.

THEN

The *statement(s)* following THEN are executed if the record is written successfully.

- ☞ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the record cannot be written. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

- ☞ For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in a WRITE statement are optional. However, it is strongly recommended that all write logic be constructed to accommodate possible unsuccessful writes. If no THEN or ELSE clause is specified, any errors during the WRITE operation are passed to the system error handling routine.

For more information, see OPEN, MATREAD, MATWRITE, READ, READV, and WRITEV.

Correct Use of WRITE Statement

```
DECLARE SUBROUTINE MSG, FMSG
@ID = 100
OPEN "CUSTOMER" TO FILE CUSTOMER THEN
  READ REC FROM FILE CUSTOMER, @ID THEN
    REC<5> = "Seattle"
    WRITE REC ON FILE CUSTOMER, @ID ELSE
      FMSG("Error writing record %1%!",@ID)
  END
END ELSE
  MSG("Cannot read record %1%!", "", "", @ID)
END
END
```

The program reads record 100 in the CUSTOMER file and writes a value to field 5.

Incorrect Use of WRITE Statement

```
WRITE ON D, "123"
```

Replacement expression is missing.

```
WRITE "OWE" ON R
```

Key expression is missing.

```
WRITE Q TO R "***"
```

Filevar is not followed by a comma.

WRITEV

The WRITEV statement updates the data content of a specified field of a record in a file. If more than one field in a record is to be updated, the commands WRITE and MATWRITE are more efficient than WRITEV.

```
WRITEV expression [ON  
TO] filevar, key, field THEN|ELSE statements
```

expression

The *expression* evaluates to the field contents that are to be written to the file.

filevar

The *filevar* refers to a file variable that has been opened in a previous OPEN statement.

✎ The WRITEV statement does not support file access using cursors.

key

The *key* specifies the key of the record to be written.

field

The *field* is the number of the field that is to be updated. The value of the *field* must be greater than zero.

THEN

The *statement(s)* following THEN are executed if the field is written successfully.

✎ For more information about the use of THEN and ELSE, see IF.

ELSE

The *statement(s)* following ELSE are executed if the field cannot be written. The STATUS() function indicates the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error. For more information, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

- For compatibility with earlier versions of Advanced Revelation, the THEN and ELSE clauses in a WRITEV statement are optional. However, it is strongly recommended that all write logic be constructed to accommodate possible unsuccessful writes. If no THEN or ELSE clause is specified, any errors during the WRITEV operation are passed to the system error handling routine.

For more information, see OPEN, MATREAD, MATWRITE, READ, READV, and WRITE.

Correct Use of WRITEV Statement

```
DECLARE SUBROUTINE FMSG
OPEN "CUSTOMER" TO FILE_CUSTOMER THEN
  @ID = 100
  READV REC FROM FILE_VAR, @ID, 5 THEN
    FIELD = "Seattle"
    WRITEV FIELD ON FILE_CUSTOMER, @ID, 5 ELSE
      FMSG()
  END
END ELSE
  FMSG("Cannot read record %1%", @ID)
END
END
```

The program reads field 5 from the CUSTOMER record 100 and assigns a value to the record variable. The new record variable is then written to field 5.

Incorrect Use of WRITEV Statement

```
WRITEV Z ON FILE, KEY ELSE STOP
```

No field expression is specified.

```
WRITEV ON FILE, KEY, 1
```

The expression is missing.

XLATE

The XLATE function is used to retrieve data from a specified field in a file.

XLATE (file, key, field, control{, expression})

file

The *file* is the name of the file where the record is to be read.

File must yield the literal name of a file. It is not a variable that has been opened to a file. The file must be in the list of files currently attached. The XLATE function opens the given file.

key

The *key* is the record key of the record that contains the specified field.

Key may contain system delimiters. One or more records may be read by one XLATE function. Multiple records are read when *key* yields a dynamic array. A *key* that yields a dynamic array must use only one level of system delimiters to separate the elements. Each element contains one record key. XLATE returns a dynamic array of fields when multiple records are read.

field

The *field* is the number of the field or name of the field in quotes (example: "COMPANY") to be extracted. If a null is specified, the entire record is returned. If a zero is specified, the record key for the record is returned.

control

The *control* must be the literal values X or C and indicates what the function should return if the record does not exist. An X indicates that a null should be returned if the record does not exist. A C will cause XLATE to return the record key if the record was not found.

expression

The *expression* indicates the number of times the system delimiters will be converted to the next lower delimiter. The default is one. The field returned from the record may contain system delimiters. If the field is a multivalued string, it becomes a sub-value string after one conversion. This feature becomes important when *key* yields a multivalued string. After the conversion each value contains a string of subvalues. A text mark (ASCII character 251) is the lowest delimiter.

STATUS()

If XLATE cannot successfully read the designated record(s), STATUS() returns information about the severity of the error, and the system variable @FILE.ERROR contains detail about the nature of the error.

Because XLATE does not support an explicit ELSE branch, @FILE.ERROR is used for this purpose. If @FILE.ERROR returns null, the XLATE function is assumed to have returned successfully.

For more information on @FILE.ERROR, see "Advanced Revelation File Access" in the chapter "Programming with R/BASIC".

Correct Use of XLATE Function

```
@ANS = XLATE("CUST",CUST_NO,3,"X")
```

The value in the third field of a CUSTOMER record is extracted. @ANS is used in dictionaries.

```
PRINT XLATE("DOS", "B:VISI83", 1, "C") [1, 10]
```

Prints the first ten characters of the DOS file VISI83, which is located on drive B.

```
PROMPT ""
```

```
KEYS = "R901" : @VM : "S433" : @VM : "R345" : @VM : "L234"
```

```
LIST DATA = XLATE("CUSTOMERS", KEYS, 7, "X")
```

```
VALUE_CNT = COUNT(LIST_DATA, @VM) + (LIST_DATA # "")
```

```
FOR VALUE_CTR = 1 TO VALUE_CNT
```

```
    PRINT "Invoice list for Customer: " : KEYS<VALUE.CTR>
```

```
    LIST_CNT = COUNT(LIST_DATA<VALUE_CTR>, @SVM) + (LIST_DATA # "")
```

```
    FOR LIST_CTR = 1 TO LIST_CNT
```

```
        PRINT LIST_DATA, VALUE_CTR, LIST_CTR
```

```
    NEXT LIST_CTR
```

```
    PRINT "_____"
```

```
    PRINT "Press [Enter] to continue:"
```

```
    INPUT PAUSE
```

```
NEXT VALUE_CTR
```

A multivalued list of keys is used to return a field from each key. Each field has its value marks converted to subvalue marks.

Incorrect Use of XLATE Function

```
Z = XLATE(2)
```

The expression is missing.



Chapter N 5

R/BASIC System Subroutines

If you are an R/BASIC programmer, you can use Advanced Revelation's windowing interface in your own applications. The routines that Advanced Revelation uses to format, display, and access data — including popups, messages, index access, and others — are available to you. In this chapter you will find information on calling the most commonly-used system subroutines from within your own R/BASIC programs.

Overview of System Subroutines	N 5 . 1
Descriptions of System Subroutines	N 5 . 3
BORDER.UP	N 5 . 5
BTREE.EXTRACT	N 5 . 6
CATALYST	N 5 . 7
COLLECT.IXVALS	N 5 . 10
User-Defined Conversions	N 5 . 12
FSMSG	N 5 . 13
INDEX.FLUSH	N 5 . 15
INIT.VIEW	N 5 . 16
INPUT.CHAR	N 5 . 18
MSG	N 5 . 19
POP.UP	N 5 . 22
REDUCE	N 5 . 28
SCRIBE	N 5 . 32
STATUP	N 5 . 41
VIDEO.RW	N 5 . 47



Overview of System Subroutines

Advanced Revelation uses system subroutines to accomplish a variety of tasks. Examples of these tasks include displaying windows, menus and popups; accepting and processing data; and maintaining and searching indexes.

The subroutines created to do these tasks are an extension of the programming tools already available in Advanced Revelation. In keeping with the “tools to build tools” philosophy of the product, this extended toolkit is available to you as a developer for your own work.

Using System Subroutines in Applications

Much of the ease of use and power of Advanced Revelation comes from the user interface it provides. The ability for users to press keys and see popups of options, related windows, TCL, and so on, streamlines considerably the development and use of even complex applications.

Many developers extend these interface features into their own applications. By calling system subroutines, you can create applications that merge seamlessly into the existing environment. Because these routines are already available, you can avoid writing custom code to accomplish these tasks.

Finding and Calling System Subroutines

You can access any system subroutine from within an R/BASIC program by using a CALL statement. The system subroutines discussed here are already available in the VERBS file — there is no requirement to load or catalog any of the programs.

As with any subroutine, these system subroutines can be declared with DECLARE statements before being called. Using DECLARE statements eliminates the need to code an explicit CALL before the subroutine name. The use of DECLARE is optional for any subroutine; it is required, however, when accessing the function POP.UP.

Additional System Information

In addition to the system subroutines themselves, Advanced Revelation accesses such information as @-variables and COMMON blocks. The @-variables are listed in the chapter “Programming with R/BASIC” under the topic “Special R/BASIC Variables”.

Most of these @-variables are dynamic arrays in which different fields have specific meanings. For instance, in the @ENVIRON.SET variable, the following fields have these meanings (there are many more than are illustrated here):

Variable	Meaning
@ENVIRON.SET<1>	Editor Ins/Ovr flag (1 = insert mode)
@ENVIRON.SET<2>	Background character (ASCII seq. no)
@ENVIRON.SET<3>	Depth of TCL stack
@ENVIRON.SET<4>	TCL stack save file
<i>(etc.)</i>	

You can read the value of these @-variables in order to get information from within an R/BASIC program about current video, color, environment, function key, or other settings. In most instances, you can also change the value of these @-variables. If you change them, though, you might consider saving their previous values before doing so and restoring the old values before exiting your program.

A Note about Constants and \$INSERT

In order to provide more readable and maintainable code, Advanced Revelation makes use of standardized system constants. These constants generally take the form of English names for flags or numeric values.

For example, it is standard in system routines for the word TRUE to be equated to the value 1, FALSE to the value 0, and OTHERWISE to the value 1. In another example, the field values for the system dynamic array @PRIORITY.INT have had English names equated for the numeric values, like this:

```
EQU GENERAL.HELP$ TO 1
EQU EXECUTE.TCL$ TO 2
EQU CAPTURE.KEY$ TO 3
(etc.)
```

In almost all cases, the constant name (for example, GENERAL.HELP\$) will have a \$ (dollar sign) appended to it in order to help distinguish it from variables in the program.

You also can use these constant definitions. Several items in the INCLUDE file (for example, WINDOW.CONSTANTS or EDIT.KEYS) consist solely of blocks of equated constants. These items can be inserted into a program at compile time with the use of the \$INSERT command.

For example, the following code fragment uses \$INSERT to include standard definitions for certain numeric values, and then references these in the program itself:

```

SUBROUTINE TEST
$INSERT INCLUDE, WINDOW.CONSTANTS
$INSERT INCLUDE, AREV.COMMON
*
* (processing here)
*
RESET = RESET.WINDOW$
RETURN

```

Note that the use of the WINDOW.CONSTANT values is optional. The next-to-last line in the above program could have been written:

```
RESET = 5
```

and the line:

```
$INSERT INCLUDE, WINDOW.CONSTANTS
```

left out. There is no functional difference between the two methods.

However, to maintain clarity and maintainability, the use of \$INSERT statements to include blocks of code is recommended. Code used in the examples will make consistent use of these blocks.

Even if you choose not to use \$INSERT to include these blocks in programs, print out the blocks and study them. The blocks provide documentation on status and flag values, dynamic array field numbers, etc., used by the system.

Descriptions of System Subroutines

This section describes system subroutines and provides brief explanations of how to call the subroutines and additional notes on their use. Full documentation for system subroutines is available in the Developer Series book entitled *System Subroutines*, available as a separate product.

Functional Guide to System Subroutines

Process	Program Name	Description
Application Processor Interface	POP.UP	Produce popup windows that offer choices or options and return user's choice(s).
(continued)		

Process	Program Name	Description
Data Entry	SCRIBE	Accept data on one or more lines, with full screen editing, data validation, and processing of reserved keystrokes.
	INPUT.CHAR	Accept a single character after screening for reserved keystrokes.
Indexing	BTREE.INVERT	Retrieve primary keys based on indexed value(s).
	COLLECT.IXVALS	Retrieve all (up to 64K) data values in a particular index.
	INDEX.FLUSH	Update (flush) pending index transactions for one or all indexes.
General	CATALYST	Process an Advanced Revelation code and command sequence.
	(Conversions)	Convert data into and out of internal and external format — callable from the R/BASIC ICONV and OCONV functions.
	REDUCE	Specify selection (reduction) criteria for a cursor.
Screen Display	BORDER.UP	Put a border around a selected portion of the screen (see also VIDEO.RW).
	FSMSG	Display an error message after a file or record access error.
	INIT.VIEW	Create a View window (with virtual space, if necessary) into which to print reports. The window can be scrolled and paged.
	MSG	Post Advanced Revelation-style messages to the screen.
	STATUP	Update the Advanced Revelation status line.
	VIDEO.RW	Clear and restore a selected portion of the screen (see also BORDER.UP).

BORDER.UP

BORDER.UP will draw borders around any area of the screen. BORDER.UP uses essentially the same arguments as VIDEO.RW, and it is common to use these two subroutines in conjunction.

```
BORDER.UP( lcol, top_row, rcol, bot_row, border, attributes )
```

lcol

An integer indicating the starting (left) column of the display area.

top_row

An integer indicating the top row of the display area.

rcol

An integer indicating the ending (right) column of the display area.

bot_row

An integer indicating the bottom row of the display area.

border

An integer indicating the type of border (1-5) to be displayed. For information on border types, see the record BORDER in SYS.HELP.

attribute

A video escape sequence to indicate the color or video attribute of the border. Attribute consists of four bytes with this format:

```
CHAR(27) : "C" : background : foreground
```

where *background* and *foreground* are single literal characters.

Correct Use of BORDER.UP

```
DECLARE SUBROUTINE BORDER.UP, INPUT.CHAR
  LCOL      = 0
  TOP_ROW   = 0
  RCOL      = 20
  BOT_ROW   = 10
  BORDER    = 5 ; * corners only
  ATTRIBUTE = CHAR(27) : "C1" ; * 1yel on dblue
  BORDER.UP(LCOL, TOP_ROW, RCOL, BOT_ROW, BORDER, ATTRIBUTE)
  STOP
```

BTREE.EXTRACT

Access to Advanced Revelation Btree indexes is normally handled transparently at the time a SELECT is generated or when a user does a [Ctrl-F10] search at a window. You can also have the system do direct lookups into Btree indexes by using a CATALYST "B" call (although this requires user interaction).

When more direct access to Btree indexes is desired, however, the system subroutine BTREE.EXTRACT can be called. BTREE.EXTRACT will look into one or more Btree indexes and return a list of record keys that correspond to data values passed to the routine.

BTREE.EXTRACT can look into more than one index in one call and can also look up more than one data value. The system allows the user to use the AND and OR logical operators to retrieve values from the Btree index. BTREE.EXTRACT only searches Advanced Revelation indexes; it will not search a foreign index in an Environmental Bond.

```
BTREE.EXTRACT( search_string, file, dictvar, keys )
```

search_string

A dynamic array of index name(s) and data value(s) to retrieve. Multiple search fields are delimited with @FM. Search values for each field are separated from the search field by @VM. *Search_string* must contain a trailing field mark.

Substring searches can be constructed using the substring search characters used with Query (see the chapter "Using Query" in *Accessing Data*). BTREE.EXTRACT supports another search operator, between (~). Between differs from Range in that it is not inclusive. For example this search string will return all zip codes between, but not including, 98100 and 98111:

```
STRING = "ZIP" : @VM : "98100~98111" : @FM
```

BTREE.EXTRACT will logically OR all of the search values for any one index. Searches using multiple fields will be logically ANDed. Just as in Query Mode, the default ANDing and ORing can be overridden with "&" and ";;".

file

The name of the file to be searched (not the file handle).

dictvar

The file variable for the dictionary of *file*.

keys

The return values (record keys) for the search string. Multiple keys are delimited with @VM. The list of keys is limited to 64K and will be truncated if longer.

Correct Use of BTREE.EXTRACT

```

* program to call BTREE.EXTRACT
* this example assumes a CUSTOMER file with Btree indexing

DECLARE SUBROUTINE MSG, INPUT.CHAR

FILE = "CUSTOMER"
OPEN "DICT.": FILE TO DICTVAR ELSE
  MSG("Can't open DICT of ":FILE,','',',')
  STOP
END

* prompt for field name to search
LOOP
  ERROR=0
  INDEX = ''
  MSG("Enter name of field to search","RC",INDEX,')
  IF INDEX THEN
    READ CONFIRM FROM DICTVAR, INDEX ELSE
      MSG(INDEX:" is an invalid field name!",','',',')
      ERROR=1
    END
  END
WHILE ERROR REPEAT

* prompt for data value(s) to search
VALUE = ''
MSG("Enter value(s) to search for, separated by ⇨
  commas", "R", VALUE, "")
IF VALUE THEN
  CONVERT ", " TO @VM IN VALUE
  SEARCH_STRING = INDEX : @VM : VALUE : @FM ; * trailing @FM
  MSG("SEARCH_STRING = ":SEARCH_STRING,','',',')

  CALL BTREE.EXTRACT(SEARCH_STRING, FILE, DICTVAR, KEYS)
  IF KEYS THEN
    MSG("The keys returned are ":KEYS,','',',')
  END ELSE
    MSG("No keys returned for value(s) ":VALUE,','',',')
  END
END
STOP

```

CATALYST

CATALYST provides a way to process codes and commands from within an R/BASIC program.

CATALYST(code, command)

code

Code contains an expression that evaluates to any valid code as documented in the chapter “Using Codes and Commands”. Multiple codes can be delimited with @SVM, and will be processed sequentially.

command

Command contains an expression that evaluates to a valid Catalyst command, as documented in the chapter “Using Codes and Commands”. Multiple commands (associated with multiple codes) can be delimited with @SVM and will be processed sequentially.

Values returned

Most CATALYST calls simply pass control to another routine. A few, however, return a value via @ANS. These are:

Code	Meaning
P	popup call (returns user’s choices)
!	system function (returns resolved function)
B	returns record key list from an index matching criteria
V	returns values from a popup of indexed values

Notes about specific calls

“W” calls The “W” call permits you to call application, system or, collector windows. When calling system windows, specify the system templates file (COMMANDS) in @-signs before the window name:

```
CALL CATALYST( "W", "@COMMANDS@RECORDCOPY")
```

This technique can be used to call any window that does not reside in the default TEMPLATES file.

When calling a collector window, a program can pass data from and to the window via the global variable @PSEUDO. A sample code fragment illustrates this:

```
* initialize PSEUDO with current rec
@PSEUDO = @RECORD
CALL CATALYST('W', 'Collector.Name')
* update current record with changes made in
* collector
@RECORD = @PSEUDO
```

"P" calls To call the system popup @ID and @IDS, the program calling CATALYST must have opened the file in question and put the file handle in the AREV.COMMON variable SRC.FILE.

Correct Use of CATALYST

```
* example showing different CATALYST calls
DECLARE SUBROUTINE MSG, CATALYST, INPUT.CHAR

* menu
CATALYST( "M", "MAIN" )
*
* window (call a command window)
CATALYST( "W", "@COMMANDS@COUNT" )
*
* popup (returns value via @ANS)
CATALYST( "P", "SYS.POPUPS*ASCII.CHART" )
IF @ANS THEN MSG("User selected option ":@ANS,'','','')
*
* different popup (show attached files) (returns value in @ANS)
* (to use @ID or @IDS, the file must have been opened to
* AREV.COMMON variable SRC.FILE)
CATALYST ( "P", "@FILE")
IF @ANS THEN MSG("User selected option ":@ANS,'','','')

* execute
CATALYST( "E", "LIST 5 VOC" )

* help message
CATALYST( "HL", "Help message called!" )

* internal function (returns value via @ANS)
CATALYST( "!", "SERIAL" )
MSG("AREV serial number = ":@ANS,'','','')
*
* btree call (returns value via @ANS)
CATALYST( "B", "CUSTOMER*COMPANY.XREF*'" )
IF @ANS THEN MSG("Btree read returned ":@ANS,'','','')
*
* dictionary call (returns value via @ANS)
@ANS = ''
OPEN 'DICT.CUSTOMER' TO @DICT THEN
  OPEN 'CUSTOMER' TO CUSTOMER.FILE THEN
    @ID = 292
    READ @RECORD FROM CUSTOMER.FILE, @ID THEN
      CATALYST( "{", "COMPANY" )
      MSG("{COMPANY} = ":@ANS,'','','')
    END
  END
END
END
```

```

* v (values) call (returns value via @ANS)
CALL CATALYST( 'V', 'CUSTOMER,COMPANY.XREF,S')
MSG("V call returned ":@ANS, ',',',',',')

* r/basic call
PGM = "PRINT @(0,23) : 'Press any key to continue:':"
PGM := "; INPUT ANS,1"
PGM := "; PRINT @(0,23) : @(-3) "
CATALYST( "C", PGM )
INPUT.CHAR(X) ; * pause
STOP

```

COLLECT.IXVALS

COLLECT.IXVALS is used to return a list of all the data values in a particular Btree index. This is different from BTREE.EXTRACT, which returns a list of primary keys associated with one or more specific data values. COLLECT.IXVALS is the basis for the CATALYST "V" call.

Because a Btree index is always maintained in sorted order, COLLECT.IXVALS will return a sorted list of data values. This list can in turn be displayed in a popup, used in a report, etc.

COLLECT.IXVALS(file_name, index_name)

file_name

The name of the data file for which the index lookup is being done.

index_name

The name of the index (field) for which data values are to be returned.

Values returned

Data values found by COLLECT.IXVALS are returned in @PSEUDO using @FM as a delimiter. The list of return values is limited to 64K; lists longer than this will be truncated.

Correct Use of COLLECT.IXVALS

```

DECLARE SUBROUTINE MSG, INPUT.CHAR

* program to call COLLECT.IXVALS
* any hits found by COLLECT.IXVALS are returned via @PSEUDO
* using @FM as the delimiter

OPEN 'FILES' TO FILES.FILE ELSE
  MSG("Unable to access FILES file!", "", "", "")
STOP

```

```

END

* prompt for file name
LOOP
  ERROR=0
  FILE_NAME = ''
  MSG("Enter file name to fetch index for","RC",FILE_NAME,')
  IF FILE_NAME THEN
    READ FILE.INFO FROM FILES.FILE, FILE_NAME ELSE
      MSG(FILE_NAME:" not a valid file!",',',',',')
      ERROR=1
    END
  END
  END
  WHILE ERROR REPEAT

* prompt for an index name
OPEN "DICT.":FILE_NAME TO FILE.DICT ELSE
  MSG("Can't open DICT of ":FILE_NAME:"!",',',',',')
  STOP
END

LOOP
  ERROR = 0
  INDEX_NAME = ''
  MSG("Enter name of index to display","R",INDEX_NAME,')
  IF INDEX_NAME THEN
    * confirm that this is a valid field
    READ CONFIRM FROM FILE.DICT, INDEX_NAME THEN
      * confirm that this is an indexed field
      CONFIRM = XLATE("!:FILE_NAME, "*INDEXES", 1, "X")
      LOCATE INDEX_NAME IN CONFIRM SETTING POS ELSE
        ERROR = 1
        MSG(INDEX_NAME:" not an index!",',',',',')
      END ; * locate
    END ELSE
      MSG(INDEX_NAME:" is an invalid field!",',',',',')
      ERROR=1
    END ; * read confirm
  END
  WHILE ERROR REPEAT

@PSEUDO = ''
CALL COLLECT.IXVALS( FILE_NAME, INDEX_NAME )
IF @PSEUDO THEN
  CONVERT @FM TO @VM IN @PSEUDO
  MSG("COLLECT.IXVALS returned|":@PSEUDO,',',',',')
END ELSE
  MSG("No values returned for COLLECT.IXVALS",',',',',')
END
STOP

```

User-Defined Conversions

In addition to the stock of conversions available to the ICONV and OCONV functions in R/BASIC, Advanced Revelation will call user-defined conversions. A user-defined conversion is a subroutine that contains all necessary logic to convert data into and out of a format required for an application. For example, you might write a function for percentages (using the '%' character), Roman numerals, etc.

```
conversion.program ( type, value, branch, return_value )
```

type

The literal ICONV or OCONV, indicating whether the conversion is an input or output conversion.

value

The data to convert

branch

Optional argument. An optional branch is passed in the ICONV and OCONV function using this syntax:

```
X = ICONV(data, "[convpgm,branch]")
```

return_value

The converted data.

STATUS()

Your conversion should set the value of STATUS() to indicate the success of the conversion operation. Possible values for STATUS() are:

- | | |
|---|---------------------------|
| 0 | Successful conversion |
| 1 | Bad data string |
| 2 | Bad conversion definition |

Correct Use of User Conversions

```
SUBROUTINE USERCONV(TYPE, VALUE, BRANCH, RETURN_VALUE)
* example of a user-defined conversion
* converts to/from decimal to percentage (nn% format)
DECLARE SUBROUTINE MSG
* this routine is called using the syntax:
*       X = ICONV(data, "[%]")
* where ICONV can also be OCONV
* (note that BRANCH is not used in this routine)

* save off data input by user
RETURN_VALUE = VALUE
```

```

* assume successful conversion
STATUS() = 0
* determine whether i/oconv requested
BEGIN CASE
  CASE TYPE = 'ICONV'
    GOSUB ICONV
  CASE TYPE = 'OCONV'
    GOSUB OCONV
  CASE 1
    BACK = ''
    STATUS()=2
END CASE
RETURN

ICONV:
* convert from the format nn% to .nn
* remove percent sign
CONVERT "%" TO '' IN RETURN_VALUE
* if not numeric, return a null, set bad status
IF NUM( RETURN_VALUE ) THEN
  RETURN_VALUE = RETURN_VALUE / 100
END ELSE
  RETURN_VALUE = ''
  STATUS() = 1
END
RETURN

OCONV:
IF NUM( RETURN_VALUE ) THEN
  RETURN_VALUE = RETURN_VALUE * 100
  RETURN_VALUE := "%"
END ELSE
  RETURN_VALUE = ''
  STATUS() = 1
END
RETURN

```

FSMSG

The routine FSMSG is used to display a file or record access error. FSMSG uses the contents of @FILE.ERROR to display the appropriate message and parameters. Optionally, FSMSG can add user-defined text or a message from the MESSAGES file to the basic error message.

- ☞ For more information about @FILE.ERROR, see “Advanced Revelation File Access” in the chapter “Programming with R/BASIC”.

```
FSMSG( { message {,values} } )
```

Using FSMSG

If a file or record access operation fails in R/BASIC, the severity of the error is indicated by the function STATUS(). In addition, the variable @FILE.ERROR is returned to your program with detail information about the error. This enables you to detect specific errors by examining the contents of @FILE.ERROR.

In many cases, however, you will only be looking for a specific error or set of errors. Rather than having to accommodate all possible error conditions, you can test for the errors you are interested in and pass the rest to FSMSG for standard handling.

message

If you do not pass any arguments to FSMSG, it will use the contents of @FILE.ERROR to generate a standard error message. However, you can pass a literal string in *message*. This string is either the key to a record in the MESSAGES file, or a literal message. If *message* is the key to a record in the MESSAGES file, the text of that message is concatenated onto the front of the standard error message before it is displayed. If *message* is not a MESSAGES key, the text is displayed as passed in the top line of the FSMSG output.

- ☛ The *message* argument in FSMSG functions exactly like the *message* argument in the system subroutine. For more information, see MSG.

values

If *message* is a key to a record in MESSAGES, *values* can be a dynamic array of values supplied to any replaceable parameters specified in *message*. The values must maintain the ordering of their respective parameters. Multiple values must be delimited by @FM, @VM, or |.

Correct Use of FSMSG

```

DECLARE SUBROUTINE MSG, FSMSG
@ID = 100
OPEN 'CUSTOMER' TO FILE CUSTOMER THEN
  READ @RECORD FROM FILE CUSTOMER, @ID THEN
    MSG("Record 100 read successfully.", "", "", "")
  END ELSE
    IF @FILE.ERROR<1> = 100 THEN
      MSG("Record 100 does not exist!", "", "", "")
    END ELSE
      FSMSG()
    END ; * read
  END ELSE
    FSMSG("Unable to open CUSTOMER file!")
  END ; * open
STOP

```


INDEX.FLUSH

In Advanced Revelation, a TCL level command (INDEX) as well as a menu option (Tools-Files-Index-Update) are provided to enable users to force a Btree index update as required.

You can have the same result within R/BASIC by calling the routine INDEX.FLUSH. INDEX.FLUSH can be configured to update one or all indexes with pending transactions.

When calling INDEX.FLUSH, you should be aware that since INDEX.FLUSH is a batch process, it will not stop for user input, unlike the normal incremental batch update. INDEX.FLUSH will only stop when the specified Btree indexes have been completely flushed.

```
INDEX.FLUSH( file_spec, index )
```

file_spec

The source datafile name, the account name for this file, and the volume label. The volume label is stored in the first field of a specified record in the VOLUMES file. The account name is stored in the third field of a specified record in the FILES file. The names must be delimited by * (asterisk). If *file_spec* is null, all indexes for all files (in attached volumes) are updated.

index

The name of the index to update. If null, all indexes for *file_spec* are updated.

Correct Use of INDEX.FLUSH

- * the example assumes a CUST file in the SYSPROG account on the
- * TUTOR volume. The field ST is assumed to be indexed using
- * a Btree index

```
DECLARE SUBROUTINE INDEX.FLUSH,MSG,FSMSG
FILE = 'CUST'
ACCOUNT = 'SYSPROG'
VOLUME = 'TUTOR'
* get the media name
OPEN 'VOLUMES' TO VOL.VAR THEN
  READ VOL.REC FROM VOL.VAR,VOLUME THEN
    MEDIA = VOL.REC<1>
  END ELSE
    FSMSG()
  END
END
FILE_SPEC = FILE:'*':ACCOUNT:'*':MEDIA
* update ST index
INDEX = 'ST'
INDEX.FLUSH(FILE_SPEC,INDEX)
* update all indexes for the file
INDEX.FLUSH(FILE_SPEC,'')
```

```
* update all indexes for all files
INDEX.FLUSH('','')
STOP
```

INIT.VIEW

The routine INIT.VIEW allows you to establish a View window (with virtual space if necessary) into which to print data. This window can be scrolled to display information that is too wide for the screen, and can be paged to see previous pages of your report.

INIT.VIEW(width)

Using INIT.VIEW

If you create a report that displays data to the screen using the PRINT statement the data is simply displayed over what is currently on the screen. If the data is too wide for the screen, it wraps to the next line. In addition, once information in your report has been replaced with a new page, you cannot recall the information just displayed.

To avoid these limitations, print to a View window. The View window captures information printed with the PRINT statement, and formats it to fit into a scrollable, pageable window. You can create a View window for your reports by calling the system subroutine INIT.VIEW.

For more information about the View window, see the chapter “Creating Reports with Advanced Revelation” in *Creating Reports*.

INIT.VIEW is designed to be used in conjunction with the paging capabilities offered by the R/BASIC statements HEADING and FOOTING. These statements invoke paging logic that is used by the View window.

INIT.VIEW should be called before you execute a HEADING or FOOTING statement in your program.

✎ INIT.VIEW should not be used with reports that are being sent to a printer.

INIT.VIEW is used in conjunction with the system variable @VIEW.MODE. Before calling INIT.VIEW, you should initialize the View window mode by setting this variable to true.

width

The width in columns of the View window. This should represent the greatest of these three values:

- the sum of the columns required to print a line of data
- the number of columns plus 2 (for margins) required to print the heading
- the number of columns plus 2 required to print the footing

If *width* is greater than 80, the View window will create virtual space for data too wide for the screen. If the View window is initialized to less than 80, no virtual space is created, and data too wide for the View window will wrap.

Correct Use of INIT.VIEW

```
@VIEW.MODE = 1
CALL INIT.VIEW( 96 )

FOR X = 1 TO 100
  PRINT "The value of X = " : X "R#3" : " " :
  PRINT STR("X",40)
NEXT
STOP
```

This program prints the numbers 1-100 plus a string of Xs into a View window 96 columns wide.

```
DECLARE SUBROUTINE MSG, INIT.VIEW, FMSG

* initialize column headings
COL_HEAD = "F1" : @FM : "F2" : @FM : "F3"
COL_LEN = 10 : @FM : 25 : @FM : 15
COLHEADING COL_HEAD
COLLENGTH COL_LEN
*-----
* calculate width of all columns for report

* add 1 extra space per column
COL_COUNT = COUNT( COL_HEAD, @FM ) + 1
VIEW_LEN = SUM( COL_LEN ) + COL_COUNT

*-----
* initialize heading and footing
HEAD = "Sample report for VOC file"
FOOT = "'L'Page 'P'"

* calculate length of heading/footing
HEAD_LEN = LEN( HEAD ) + 2 ; * adding 2 accounts for margin
FOOT_LEN = LEN( FOOT ) + 2

*-----
* use widest length to calculate width of View window
IF HEAD_LEN > VIEW_LEN THEN VIEW_LEN = HEAD_LEN
IF FOOT_LEN > VIEW_LEN THEN VIEW_LEN = FOOT_LEN

*-----
* initialize View window
@VIEW.MODE = 1
CALL INIT.VIEW( VIEW_LEN )

*-----
HEADING HEAD
FOOTING FOOT
```

```

*-----
OPEN 'VOC' TO FILE.VOC ELSE
  FMSG()
  STOP
END
SELECT FILE.VOC
DONE = 0
LOOP
  READNEXT @ID ELSE DONE = 1
UNTIL DONE DO
  READ @RECORD FROM FILE.VOC, @ID THEN
    PRINT @RECORD<1> "L#10" : " " :
    PRINT @RECORD<2> "L#25" : " " :
    PRINT @RECORD<3> "L#15" : " " :
  END
REPEAT
STOP

```

INPUT.CHAR

INPUT.CHAR is the standard input routine for all Advanced Revelation processes. INPUT.CHAR waits for, accepts and processes one keystroke at a time. Reserved keystrokes (those defined in @PRIORITY.INT such as [F5]) are not passed through, but are processed as defined in the system. In addition, certain user environment restrictions may apply (example: TCL disabled). INPUT.CHAR is also the basis for the indexing timeout loop, used to determine when background indexing is to occur.

INPUT.CHAR (key)

key

Returns with a keystroke in it. If the keystroke is an extended character (example: [F1]), it is returned as two ASCII characters; the first will always be an ASCII zero.

Correct Use of INPUT.CHAR

```

DECLARE SUBROUTINE INPUT.CHAR, MSG
EQU ESC TO CHAR(27)
MESSAGE = "Determine ASCII sequence of a keystroke."
MESSAGE := "|Press any key ( [Esc] to quit )"
IMAGE = ''
MSG(MESSAGE, "UB", IMAGE, "") ; * message up till DB call
LOOP
  INPUT.CHAR(KEY)
UNTIL KEY = ESC
  CHAR1 = SEQ( KEY[1,1] )
  CHAR2 = SEQ( KEY[2,1] )
  IF CHAR2 THEN
    STRING = CHAR1:",":CHAR2
  
```

```

END ELSE
  STRING = CHAR1
END
MESSAGE = "ASCII sequence = "
MESSAGE := "|" : STRING
MSG(MESSAGE, '', '', '')
REPEAT
MSG(' ', 'DB', IMAGE, '')
STOP

```

MSG

MSG is used to post messages to the screen. You can create your own messages using MSG or can call system messages stored in the MESSAGES file. MSG can also be used to accept input from the user.

- ☛ New messages can be added to the MESSAGES file by using the MESSAGE window. Enter CW MESSAGE at TCL to display this window.

MSG(message, map, response, values)

message

The key to a message record in the MESSAGES file, or developer-specified message text. *Message* can include replaceable parameters that are filled when the message is displayed. The following are replaceable parameters:

Parameter	Meaning
%n%	replaced with element <i>n</i> in <i>values</i>
%D%	replaced with the current date
%G%	replaces with all values in <i>values</i>
%T%	replaced with the current time
%B%	rings the bell
%Cn%	replaced with ASCII character <i>n</i>

map

A message record. Each field is delimited with @FM. The record can be read from file or created dynamically. (For information about the layout of the message record, see the dictionary of the MESSAGES file.) You can pass an entire message record, or you can pass only specific fields. These fields override the default values for the fields as assumed by MSG or as established in the message record referenced by *message*.

Information about messages types can be found in the system popup MESSAGE.TYPES. The following is an abbreviated list of available messages types:

Type	Meaning
A	The message is displayed until a key is pressed.
R/RC/RE	The message displays and prompts the user for a response.
U/UB	The message is displayed and the calling program continues. UB displays a message with a border.
D/DB	A message displayed earlier with U or UB is taken down. DB removes the border specified in UB.
Tn	The message is displayed for <i>n</i> seconds.

response

Used primarily by MSG to return user input or store a previous display image (see below). If specified, *response* must be an initialized variable. This value may serve as the default for those messages accepting user input. The manner in which a *response* value is processed is determined by the message type specified in *map*. The following Message record type specifiers return a value in *response*:

MSG Type	Response
U, UB	Saves the current display image in <i>response</i> , then displays <i>message</i> .
R	Displays <i>message</i> and an edit line and returns any input data. If <i>response</i> is specified, it is displayed at the edit line and serves as the default value returned. The user may edit the data displayed.
RC	The same as the R type, except all lowercase characters are converted to uppercase when the data is returned.
RE	The same as the R type, except the ASCII code for the escape character will be returned if pressed.
RCE	The same as RC, but returns an [Esc] character (ASCII 27) if the [Esc] key has been pressed.
RI	Displays <i>message</i> and an input line (see R/BASIC INPUT) and returns any input data.

values

A dynamic array of values supplied to any replaceable parameters specified in *message*. The values must maintain the ordering of their respective parameters. Multiple values must be delimited by @FM, @VM, or |.

Values returned

Depending upon the message record type specified, MSG may return a value in the *response* argument list parameter (see above).

Correct Use of MSG

```
* illustrates different types of MSG calls

* this routine does a variety of calls to the MSG routine to
* illustrate how each of the different types is done

DECLARE SUBROUTINE MSG, INPUT.CHAR

* save screen image
CALL VIDEO.RW(0,0,@CRTWIDE-1, @CRTHIGH-1, 'R', IMAGE)
CALL VIDEO.RW(0,0,@CRTWIDE-1, @CRTHIGH-1, 'C', '')

* call an existing message in the MESSAGES file.
MSG("B213",'A','','','')

* create a message text and display it as a default type
MSG("Custom text",'','','','')

* display a message and solicit a response
RESP = '' ; * initialize variable to hold response
MSG("Enter any character:","RC",RESP,'')

* parameter substitution msg
* (note use of "|" to separate lines)
MSG("The character you entered was | %1% ","','','',RESP)

* put up a message, then leave it up as a process is
* accomplished
IMAGE2 = '' ; * initialize a variable to hold the screen image
MSG('Processing ...', 'UB', IMAGE2, '')

* do some processing here
FOR CTR = 1 TO 25
  PRINT @(0,22) :
  MSG("Processing ":CTR,'H','','','')
NEXT CTR

* take down first message & restore screen image
MSG( '', 'DB', IMAGE2, '')

* timed message
MSG("This message is up for 3 seconds.,"T3",'','','')
```

```

* message created with entire message map intact
* (the entire map is created in the program and passed to MSG.
* Refer to the DICT of MESSAGES for information about
* each element)
MAP = ''
MAP<1> = 'A'      ; * type
MAP<2> = ''       ; * buffering (B=buffer, R=read & buffer @ logon)
MAP<3> = 3        ; * column
MAP<4> = 5        ; * row
MAP<5> = ''       ; * video attributes ('' = default)
MAP<6> = "L"      ; * justification
MAP<7> = 30       ; * width
MAP<8> = 1        ; * border type (1-5)
MAP<9> = ''       ; * unused
MAP<10> = ''      ; * unused
MAP<11> = "An example|of a fully coded|message call" ; * script
MAP<12> = ''      ; * character count limit for "R" messages
MSG('',MAP,'','',)

* restore screen image
CALL VIDEO.RW(0,0,@CRTWIDE-1, @CRTHIGH-1, 'W', IMAGE)
STOP

```

POP.UP

POP.UP is an external function that creates and displays a popup, returning the user-selected popup display value(s).

```
POP.UP(col, row, file, display, format, mode, select, ⇐
      title, attributes, help, coordinates, type)
```

col

An integer indicating the starting display column for the popup.

row

An integer indicating the starting display row for the popup.

file

The file providing the data to *display* for display in the popup. If *file* is prefixed with @ (at-sign), POP.UP will attempt to open the file, otherwise *file* must be a file variable. If *file* is null, POP.UP assumes the literal data is to be found in *display*.

display

Display determines how the *file* data is to be interpreted and displayed, providing the literal data if *file* is null. The format of *display* is dependent upon the *mode*

parameter (see below). The following is a list of the modes and corresponding values expected in *display*:

Mode	Value
T	If <i>file</i> is specified, <i>display</i> must be a dynamic array of record keys (delimited by @FM). Each record key is displayed in the first data column of the popup. If <i>display</i> is null, the current select list is used If there is no select list and QUICKDEX is installed for <i>file</i>, the records are listed in QUICKDEX order. If <i>file</i> is null and the popup is unformatted (see <i>format</i> below), <i>display</i> should be a dynamic array of the literal data to display. Display rows correspond to records (@RM delimited) and columns to record fields (@FM delimited).
R	If <i>file</i> is specified, <i>display</i> must be a single record key. The fields of the specified record are displayed in the first data column of the popup. If <i>file</i> is null, <i>display</i> is expected to be a dynamic array of the literal data to display. Display rows correspond to fields (@FM delimited) and columns to field values (@VM delimited). Note that if more than one value is passed in <i>display</i>, the <i>file</i> specification is ignored, and <i>display</i> is treated as literal data.
F	If <i>file</i> is specified, <i>display</i> must be a single record key and field number (delimited by @FM) indicating the field to be displayed in the first data column of the popup. If <i>file</i> is null, <i>display</i> is expected to be a dynamic array of literal data. Display rows correspond to field values (@VM delimited) and columns to subvalues (@SVM delimited).

format

Determines the format for each column of the popup. This may be a single integer indicating the display width of the (unformatted) popup, or formatting criteria concatenated by : (colon). Unspecified parameters must be bracketed by concatenation marks (an example format specification might be: 2:25:C::Company). Formatting specifications for multiple data columns are delimited by \ (backslash).

The following ordered parameters may be specified for each data column:

Parameter	Value
field	Field number or name corresponding to the <i>file</i> or <i>display</i> field. Note that if <i>mode</i> is T, a value of 0 indicates the key field.
width	Data column width. Must be an integer indicating total width.
just	Justification of displayed data. May be C (centered), L (left), or R (right). If null, the default is left.
oconv	Output conversion for the data column. Must be a conversion pattern acceptable to OCONV (see R/BASIC OCONV). If null, no conversions will occur.
heading	Data column heading. If null, the column will be unheaded.

mode

Determines the interpretation of *file* data (or *display* data, if *file* is null). The following are the modes available (defaults to R):

Mode	Meaning
T	Table
TS	Table. Saves the current select list before normal T mode operation. The select list is restored when POP.UP returns.
R	Record
F	Field

select

Determines the user selection process for the popup. If *type* is R (read only), *select* specifications have no effect. The following is a list of values that may be specified (defaults to 0):

Value	Selection
null or 0	Single selection. The user must press [Enter] to select (and save) the selection.
1	Multiple selection. The user must press [Enter] to select, and [F9] to save the selections.

(continued)

Value	Selection
G	Group selection. This is a select array of the following fields (delimited by @FM): <1> G (the select type) <2> Data column to group <3> Starting group number. This number is displayed in the field 2 column specified above as the user selections are grouped.
O	Ordered multiple selection. A final data column is included. Based on the user selection, a number is displayed in each row, ordering the selections.
S	Sort data rows. The user must press [Enter] to select, [Up] or [Dn] to move, and [Enter] again to insert at a new location. The resulting order must be saved by pressing [F9]. This is specified as a select array of the following fields (delimited by @FM): <1> S (the <i>select</i> type) <2> Data column to sort <3> Starting row to sort (defaults to 1) <4> Ending row to sort
T	Toggled selection. The user selection is determined by the indicator (fields 4 and 5 in the <i>select</i> array) displayed in the specified data column when the selection(s) is saved (by pressing [F9]). The indicators are toggled by pressing [Enter]. This is specified as a <i>select</i> array of the following fields (delimited by @FM): <1> T (the <i>select</i> type) <2> The field number, in the selection returned, of the indicator (defaults to 2). <3> The data column in which to display the toggled indicators (defaults to 2). <4> The indicator to accept the selection (defaults to "Yes"). <5> The indicator to reject the selection (no default). This value is the toggled alternative to <4>.

title

Title of popup. Displayed at the top row. Multiple title lines must be delimited by | or @FM. Title justification defaults to centered. This may be overridden by appending the literal value L (left) or R (right) to the title data (delimited by @SVM). Note that this setting will affect only the last line of a multiline title.

attributes

An Advanced Revelation escape sequence indicating a video attribute for the popup border. The system variables @PW or @ENVIRON.SET define further video attributes that may be set. *Attributes* defaults to the value of @PW, an array of popup attributes.

help

Determines the process for displaying help information when the user presses the [F1] (Context) help key. The following formats may be used:

Format	Operation
Key	Calls CATALYST with an HF code and the specified help message record key for the SYS.HELP or HELP file (SYS.HELP is searched first) as the associated command.
code, command	Calls CATALYST with the specified (help) code and command as arguments.

coordinates

The data column and row to be returned for the selection. This is indicated in the popup by the cursor or highlight position. The coordinates will have effect only if *type* is not specified. The values are integers:

- <1> Data column (defaults to 1).
- <2> Data row. Effective only for multivalued fields, otherwise the coordinates <1> value is used (defaults to coordinates<1>).

type

The type of data (if any) to be returned for each user selection. Note that *type* specifications will override any conflicting specifications in *select* or *coordinates*. If *type* is null, *coordinates* determines the data column and row returned for a selection. The following types may be specified:

Type	Value Returned
K	Record key(s) of selected data.
R	Read only. No value returned..
P	Row number(s) of selection(s).
null	Data at selected row (complete display row).

Values returned

POP.UP returns the popup display value(s) selected by the user and of the *type* specified.

Correct Use of POP.UP

```

DECLARE SUBROUTINE MSG
DECLARE FUNCTION POP.UP
COL          = 0
ROW          = 0
FILE        = ''
DISPLAY     = "Edit File"   : @VM : "Name of file to edit"
DISPLAY<-1> = "Delete File" : @VM : "Name of file to delete"
FORMAT      = 30
MODE        = 'R'
SELECT      = ''
TITLE       = 'UTILITIES|FILE OPERATIONS'
ATTRIBUTES  = ''
HELP        = ''
COORDINATES = 1
TYPE        = ''

```

```

* display a two-column, single selection, unformatted popup,
* returning the user selection in @ANS
@ANS = POP.UP(COL,ROW,FILE,DISPLAY,FORMAT,MODE,SELECT, ⇨
              TITLE,ATTRIBUTES,HELP,COORDINATES,TYPE)
MSG('The user selection:|':QUOTE(@ANS),' ',' ',' ')

```

```

* display a two-column, multiple selection, formatted popup, and
* specify the type of user-selected data to be returned in @ANS
DISPLAY     = "Edit File"   : @SVM : "Name of file to edit"
DISPLAY<1,-1> = "Delete File" : @SVM : "Name of file to delete"
FORMAT      = '1:12:L::Action\2:25:L::Description'
SELECT      = '1'
MODE        = 'F'
TYPE        = 'P'
@ANS = POP.UP(COL,ROW,FILE,DISPLAY,FORMAT,MODE,SELECT, ⇨
              TITLE,ATTRIBUTES,HELP,COORDINATES,TYPE)
MSG('The user selection:|':QUOTE(@ANS),' ',' ',' ')

```

```

* display a three-column, group selection, formatted popup, and
* specify a help message record
FORMAT      = "1:12:L::Action\2:20:L::Description\3:9:C::Choice"
SELECT      = 'G':@FM:3:@FM:1
HELP        = "AW*POPUP.DOC"
@ANS = POP.UP(COL,ROW,FILE,DISPLAY,FORMAT,MODE,SELECT, ⇨
              TITLE,ATTRIBUTES,HELP,COORDINATES,TYPE)
MSG('The user selection:|':QUOTE(@ANS),' ',' ',' ')

```

```

* display a three-column, toggled selection, formatted popup.
* a sample PRODUCT file is assumed
FILE        = '@PRODUCT'
TITLE       = 'ORDERS'
DISPLAY     = ""
FORMAT      = "0:5:L::Code\1:20:L::Description\10:6:C::Order"
MODE        = 'T'
SELECT      = 'T':@FM:1:@FM:3:@FM:'' :@FM:'No'

```

N 5

R/BASIC System Subroutines

```

@ANS = POP.UP (COL,ROW,FILE,DISPLAY,FORMAT,MODE,SELECT, ⇨
    TITLE,ATTRIBUTES,HELP,COORDINATES,TYPE)
MSG('The user selection:|':QUOTE(@ANS),' ',' ',' ')

* display a three-column, sorted selection, formatted popup,
* specifying the display attributes (see @PW) and the type of
* user-selected data to be returned in @ANS.
$INSERT INCLUDE,WINDOW.POINTERS
DISPLAY      = "Edit File"      : @VM : "Name of file to edit"
DISPLAY<-1>   = "Delete File"   : @VM : "Name of file to delete"
FORMAT       = "1:11:L::Function\2:25:L::Description"
MODE         = 'R'
SELECT       = 'S':@FM:1:@FM:1:@FM:''
TYPE        = ''
* set video attributes for the popup
ESC = CHAR(27)
ATTRIBUTES<CBORDER> = ESC:'C07' ; * black
ATTRIBUTES<CTITLE>   = ESC:'C00' ; * highlight
ATTRIBUTES<CBAR>     = ESC:'C00' ; * highlight
ATTRIBUTES<CBORDER.TYPE> = 1
@ANS = POP.UP (COL,ROW,FILE,DISPLAY,FORMAT,MODE,SELECT, ⇨
    TITLE,ATTRIBUTES,HELP,COORDINATES,TYPE)
MSG('The user selection:|':QUOTE(@ANS),' ',' ',' ')
STOP

```

REDUCE

REDUCE allows you to specify selection (reduction) criteria for a cursor. The reduction parameters that you specify are activated when a SELECT ... BY statement is executed.

```

REDUCE(reduce_script, sort_list, mode, file_name, ⇨
    cursorvar, flag)

```

reduce_script

An expression containing the formatted reduction criteria. The information passed in *reduce_script* must be formatted as follows:

- All field names must be enclosed in braces ({ }).
- Field names must not have spaces in them.
- All constants must be on the right side of the comparison operator and must be enclosed in either single or double quotes.
- All comparison operators must be reduced to the primitive set of operators and must be separated by spaces (see chart below).

Operator	Primitive
EQUAL TO	EQ
LESS THAN	LT
GREATER THAN	GT
LESS THAN OR EQUAL TO	LE
GREATER THAN OR EQUAL TO	GE
STARTING WITH	[
ENDING WITH	]
CONTAINING	[]
MATCHES	MATCH
RANGE	FROM TO
SUNDAY	SUN
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
YESTERDAY	YST
TOMORROW	TOM
NEXT	NEXT
LAST	LAST

Example assignments for REDUCE_SCRIPT:

```
REDUCE_SCRIPT = "WITH {DATE} EQ '9-23-59'"

REDUCE_SCRIPT = "WITH {CITY} EQ 'SEATTLE' ⇨
OR WITH {STATE} EQ 'CA'"

REDUCE_SCRIPT = 'WITH {GRADE_POINT} FROM ⇨
"2.0" TO "4.0"'

REDUCE_SCRIPT = 'WITH {MAJOR} EQ ' : ⇨
QUOTE(MAJOR_VAR)

REDUCE_SCRIPT := ' AND WITH {MINOR} EQ ' : ⇨
QUOTE(MINOR_VAR)

REDUCE_SCRIPT = 'WITH {INV_DATE} LT ' : ⇨
QUOTE( DATE() - 30 ) : ' OR WITH ⇨
{INVOICE_AMT} GE "1000"'

REDUCE_SCRIPT = 'WITH {PHONE} MATCH "3N-4N"'
```

sort_list

An expression containing the sort criteria. The sort criteria requires the same format as when used in SELECT ... BY. If *sort_list* is preceded with a percentage sign (%), the status bar will be displayed while the records are being sorted.

If the program intends to sort, both the call to REDUCE and the SELECT statement must use the same *sort_list*.

Example assignments for *sort_list*:

```
SORT_LIST = "CITY"

SORT_LIST = "ST" : @FM : "CITY"

SORT_LIST = "#INV_TOTAL"

SORT_LIST = "ST" : @FM : "#INV_TOTAL"
```

mode

An expression that resolves to one of three values:

- 0 Indicates that these are new reduction criteria for an existing cursor.
- 1 Indicates that the subroutine should return the results of the reduction in the next available cursor.
- 2 Indicates that these are additional criteria for an existing cursor. The new criteria will be joined with a logical AND to any previous criteria.

file_name

An expression resolving to the literal file name (not a file variable).

cursorvar

A variable representing a cursor already initialized (MODES 0 and 2) or a variable that will represent a cursor to be initialized (MODE 1). An explicit cursor number may also be passed.

flag

A variable that will contain 1 (true) if the subroutine was successful or 0 (false) if unsuccessful.

Correct Use of REDUCE

```
DECLARE SUBROUTINE REDUCE,MSG,FSMSG
*
FILE_NAME = "CUSTOMER"
FLAG      = ""
CURSORVAR = ""
MODE      = 1

* '%' means display status bar if appropriate
SORT_LIST = "%ST"
REDUCE_SCRIPT = "WITH {ST} EQ 'WA' OR WITH {ST} EQ 'CA'"
*
```



```

REDUCE(REDUCE_SCRIPT, SORT_LIST, MODE, FILE_NAME, ⇨
      CURSORVAR, FLAG)
*
IF FLAG ELSE
  MSG("Failure in REDUCE","", "", "")
  STOP
END
*
SELECT FILE_NAME BY SORT_LIST USING CURSORVAR ELSE
  FMSG()
  STOP
END
*
OPEN FILE_NAME TO FILE_VAR ELSE
  MSG("Can't find %1%. ", "", "", FILE_NAME)
  STOP
END
*
FLAG = 0
FIRST_KEY = ""
LOOP
  READNEXT KEY USING CURSORVAR BY AN ELSE NULL
  IF NOT(FIRST_KEY) THEN
    FIRST_KEY = KEY
  END ELSE
    IF FIRST_KEY = KEY THEN FLAG = 1
  END
UNTIL FLAG = 1
  READ REC FROM FILE_VAR, KEY ELSE
    MSG("%1% is missing", "", "", KEY)
  END
  MSG("Key = %1%|State = %2%", "T1", "", KEY:@VM:REC<5>)
REPEAT

* resetting mode indicates that these are additional
* criteria for an existing cursor.
MODE = 0
SORT_LIST = "%ZIP"
REDUCE_SCRIPT = 'WITH {ST} EQ "CA"'
REDUCE(REDUCE_SCRIPT, SORT_LIST, MODE, FILE_NAME, ⇨
      CURSORVAR, FLAG)
*
IF FLAG ELSE
  MSG("Failure in REDUCE","", "", "")
  STOP
END
SELECT FILE_NAME BY SORT_LIST USING CURSORVAR ELSE
  FMSG()
  STOP
END

FLAG = 0
FIRST_KEY = ""
LOOP

```

```

    READNEXT KEY USING CURSORVAR BY AN ELSE NULL
    IF NOT(FIRST_KEY) THEN
        FIRST_KEY = KEY
    END ELSE
        IF FIRST_KEY = KEY THEN FLAG = 1
    END
    UNTIL FLAG = 1
    READ REC FROM FILE_VAR,KEY ELSE
        MSG("%1% is missing","", "",KEY)
    END
    MESSAGE = ""
    MESSAGE<-1> = " Key = %1% "
    MESSAGE<-1> = " St  = %2% "
    MESSAGE<-1> = " Zip = %3% "

    MSG(MESSAGE, "T1", "", KEY:@VM:REC<5> :@VM:REC<6>)
    REPEAT
    STOP

```

SCRIBE

SCRIBE is Advanced Revelation's editor. It is the same editor called by the full-screen editor, as well as that called at each prompt of a window. In effect, SCRIBE functions like the INPUT statement in R/BASIC. However, it offers much more power and flexibility in the form of cursor control, multi-line entry, data validation and masking, and compatibility with Advanced Revelation reserved keystrokes (such as [F5] for TCL). SCRIBE can store its environment, which can be restored for subsequent calls.

SCRIBE can handle input and editing of one or more lines. If more than one line is being displayed, full-screen editing capability is available in the form of up and down arrow keys, page up and page down, etc. SCRIBE will automatically scroll data through the window or line size defined for it by the calling parameters.

```

    SCRIBE( inval, protect, exit_key, except_key, col, row, ⇨
           displen, color, flags, change, lines, exit, inpat, ⇨
           output, just, delim, mask, linelen, state )

```

inval

The data to be displayed and edited. SCRIBE also returns the edited data here. In multiline mode (*lines* is greater than 1), the lines of text returned in *inval* will be delimited by the character specified in *delim*.

protect

If true, this flag prohibits the editing of *inval* data. A system message will be generated if editing is attempted. *Protect* defaults to false.

exit_key

The key or key sequence that will exit SCRIBE whenever editing is enabled. Multiple *exit_keys* must be separated by | or @FM. In addition, exiting will under the following conditions:

[↑] is pressed at the top of an edit window or [↓] at the bottom.

[Enter] is pressed and *lines* is 0 (single line edit).

except_key

The key or key sequence that will exit SCRIBE whenever editing is disabled. Multiple *except_keys* must be separated by | or @FM. In addition, exiting will occur if [↑] is pressed at the top of an edit window or [↓] at the bottom.

col

An integer indicating the starting display column of the edit environment.

row

An integer indicating the top display row of the edit environment.

displen

An integer indicating the display length, in columns, of the edit environment. *Displen* must be specified. If the length of the text entered exceeds *displen*, the text is scrolled horizontally.

color

An array of Advanced Revelation escape sequences indicating the video attributes for the edit window. The attributes default to the settings in @ENVIRON.SET. The following *color* fields may be set:

Field	Description
<1>	An escape sequence setting the video attributes of all text displayed.
<2>	An escape sequence setting the video attributes for any defined block of text.
<3>	An escape sequence setting the video attributes for text at the current line.

flags

Dynamic array of SCRIBE environment parameters. The following are the definitions for the *flags* fields (these are also defined in the SCRIBE.FLAGS.CONSTANTS record):

```

EQU ST.CHAR$      TO 1
EQU NOPAINT$      TO 2
EQU UPPER.CASE$   TO 3
EQU EDIT.NAME$    TO 4
EQU FORCE.EDIT$    TO 5
EQU REPAINT$      TO 6
EQU NOSUBVALUES$  TO 7
EQU SAVE.STATE$   TO 8
EQU NOZOOM$       TO 9
EQU NO.ORIG.VAL$  TO 10
EQU EDITOR.FLAG$  TO 11
EQU KEEP.STATUS$  TO 12

```

The requirements for the field values are described below:

Field	Description
<ST.CHAR\$>	Pre-process commands. This must be the ASCII code for the keystroke or keystroke sequence that calls the desired process. The function key [F1], for example, would be passed as CHAR(0):CHAR(59). Multiple keystroke sequences must be delimited by or @FM. The associated operation will be performed after <i>inval</i> is displayed and prior to the processing of the keystroke buffer (the value of @DATA).
<NOPAINT\$>	Suppress display of text. If true, text is not initially displayed.
<UPPER.CASE\$>	Case conversion. If true, all text is converted to uppercase.
<EDIT.NAME\$>	Field name. The name of the field being edited (replaces the title in the Zoom window).
<FORCE.EDIT\$>	Insert. If true, SCRIBE will be activated in Insert mode.
<REPAINT\$>	Repaint. If true, the current line is displayed upon return to the SCRIBE edit session (saved in <i>state</i> <SS.LAST.VAL>). If false (the default value), the current line is not displayed. If the value is 2, the entire window is displayed (the value is then reset to false). This field value may be used in association with <i>state</i> specifications to redisplay text.
<NOSUB-VALUES\$>	Subvalue edit. If true, subvalue editing is disabled.

Field	Description
<SAVE.STATE\$>	Save current state. If true, the current edit session parameters are automatically saved in <i>state</i> . SCRIBE will then update these parameters during the edit process.
<NOZOOM\$>	If true, Zoom mode is disabled.
<NO.ORIG.-VALUES\$>	Suppress save of <i>inval</i> data. If false, and if <i>state</i> <SS.ORIG.VAL> is null, a copy of the original <i>inval</i> data is saved in <i>state</i> (defaults to false).
<EDITOR.FLAG\$>	A Boolean flag. A value of true indicates a recursive call to SCRIBE.
<KEEP.STATUS\$>	A Boolean flag. If true, the status line is not overwritten unless a key is pressed.

change

A flag used by SCRIBE to indicate whether edit changes occurred to *inval* data. False is returned if no changes were made, otherwise true.

lines

Indicates single or multiple line entry modes. The following listed values are operable (defaults to 1):

Value	Operation
0	Single line edit.
1	Single line scroll for multi-line edit. A single line is displayed, the display scrolling vertically to the next line.
<i>n</i>	Full screen edit (where <i>n</i> is an integer indicating the number of display rows in the window). <i>Delim</i> must be specified.

key

Used by SCRIBE to return the ASCII code for the exit key used.

inpat

Determines input pattern data validation and conversion. Syntactically as well as operationally, this is exactly the same as an In Pattern operation in the Paint window. Multiple *inpat* statements are processed as logical OR values and must be delimited by @VM or |. Inpat is applied to each line if *lines* specifies multi-line or single line scroll mode.

outpat

Determines output pattern conversion (applied line by line). Syntactically as well as operationally, this is exactly the same as an Out Pattern operation in the Paint window (see the Paint documentation for more details).

just

Determines display line justification. The possible values are:

Value	Operation
L	Left justified
R	Right justified
C	Center justified
T	Text (text lines will wrap). <i>Delim</i> delimits each line.
TN	Text nonjustified (text lines will wrap).

delim

The character used to delimit multiple lines of data in SCRIBE. This may be an accepted Advanced Revelation delimiter (e.g. @FM), an ASCII code, or a literal character. If *delim* is null, only single line text entry is enabled, regardless of any other specifications (this will cause the display of *inval* data to be suppressed if *lines* specifies multiline mode).

mask

Defines an input mask. This indicates to the user the length of data expected to be input. The mask is displayed at the current edit line. If *mask* is a single character, it is repeated for the length of the line.

linelen

Defines the maximum length of the edit line (defaults to 64K). If * (asterisk) is specified, the value of *displen* is used. A carriage return is generated at the end of the line, unless *just* is T or TN.

state

A dynamic array (delimited with @RM) of edit session parameters. These parameters are saved in *state*, allowing recall of a specified edit environment. If *flags*<SAVE.STATE\$> is set to true, SCRIBE saves the current edit environment (as opposed to a specified edit environment) in *state*, automatically maintaining (updating) these values during the edit session. Any existing *state* parameters are overwritten.

Note that the *state* specifications save but do not redisplay the edit session data. This data may be redisplayed by setting the *flags*<REPAINT\$> field (see above).

If *state* is null, any previous *state* settings are cleared. The following fields are defined for *state* (these are also defined in the SCRIBE.STATE.CONSTANTS record):

```

EQU SS.ORIG.VAL      TO 1
EQU SS.LAST.VAL      TO 2
EQU SS.TOP           TO 3
EQU SS.MVD           TO 4
EQU SS.START         TO 5
EQU SS.POS           TO 6
EQU SS.EDITOR.ON     TO 7
EQU SS.UNUSED1       TO 8
EQU SS.FIND          TO 9
EQU SS.REPLACE       TO 10
EQU SS.STATE         TO 11
EQU SS.BLOCK.START   TO 12
EQU SS.BLOCK.END     TO 13

```

The requirements for the field values are described below:

Field	Description
<SS.ORIG.VAL>	Original <i>inval</i> data (<i>flags</i> <SAVE.STATE\$> must be false).
<SS.LAST.VAL>	Contents of current display line. This must be explicitly appended to the ending <i>state</i> field for redisplay (delimited by @RM).
<SS.TOP>	Top display row of window.
<SS.MVD>	Display row of current line.
<SS.START>	Starting display column.
<SS.POS>	Cursor location (display column) in current line.
<SS.EDITOR.ON>	Edit flag. A setting of true enables editing; false disables it.
<SS.UNUSED1>	Reserved for future use.
<SS.FIND>	Current search string.
<SS.REPLACE>	Current replacement string.
<SS.STATE>	Changes in edit status. Note those values passed to SCRIBE and those set by SCRIBE. The status is indicated by the following values:
null	No changes (set by SCRIBE).
1	Changes in current row preceding <i>inpat</i> operations (set by SCRIBE).

(continued)

Field	Description
2	Process current row through <i>inpat</i> (passed to SCRIBE).
3	Process full edit session through <i>inpat</i> (passed to SCRIBE).
<SS.BLOCK. START>	Starting row of current block.
<SS.BLOCK.END>	Ending row of current block.

Values Returned

The following is a list of the argument list parameters in which SCRIBE passes values, a description of the value returned, and any conditions governing the return of this value:

Parameter	Value Description	Conditions
<i>inval</i>	Edited <i>inval</i> data	None. Always returned.
<i>key</i>	The key pressed to exit SCRIBE	None. Always returned.
<i>change</i>	True or false (1 or 0)	None. Always returned.
<i>state</i>	Current edit session parameters	SCRIBE must have been called with <i>flags</i> <SAVE.STATE\$> set to true.
	Specified edit session parameters	SCRIBE must have been called with <i>flags</i> <SAVE.STATE\$> set to false and the specified edit session parameters set in <i>state</i> .
	The original (unedited) value of <i>inval</i>	SCRIBE must have been called with <i>flags</i> <NO.ORIG.VAL\$> set to false and <i>state</i> as null.

Correct Use of SCRIBE

```
* this program prompts the user for information about the
* location of a window on the screen, and then calls SCRIBE to
* edit data within the window. When SCRIBE is exited, the
* program reports on the return value of certain paramters.
*
* The program loops through and re-calls SCRIBE until the user
* chooses to quit. At each iteration, the user can re-
* initialize the data passed to SCRIBE.
*
$INSERT INCLUDE, SCRIBE.FLAGS.CONSTANTS
```



```

$INSERT INCLUDE, SCRIBE.STATE.CONSTANTS
$INSERT INCLUDE, LOGICAL
$INSERT INCLUDE, WINDOW.POINTERS

DECLARE SUBROUTINE MSG, VIDEO.RW, SCRIBE, BORDER.UP, INPUT.CHAR

EQU NULL          TO ""
EQU F9             TO CHAR(0):CHAR(67)
EQU ESC           TO CHAR(27)
EQU CURSOR.DOWN   TO CHAR(0):CHAR(80)

* use VIDEO.RW to save and clear screen area
VIDEO.RW(0,0,@CRTWIDE-1,@CRTHIGH-1,'R',IMAGE1)
VIDEO.RW(0,0,@CRTWIDE-1,@CRTHIGH-1,'C',"") ; * clear window

* create background (read background char from current
* environment)
BCHAR = CHAR(@ENVIRON.SET<2>)
BACKGROUND = 5 ; * background video attribute
FOREGROUND = 14 ; * foreground video attribute
COLOR = (BACKGROUND * 16) + FOREGROUND ; * color formula
COLOR = CHAR(COLOR)
ATTRIBUTES = BCHAR:COLOR
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'C',ATTRIBUTES)

* prompt for size and location of box
LEFT=10
MSG("Enter leftmost column location","R",LEFT,'')
TOP=5
MSG("Enter topmost row location","R",TOP,'')
WIDTH=20
MSG("Enter width of window","R",WIDTH,'')
DEPTH = 8
MSG("Enter depth of window","R",DEPTH,'')
RIGHT = LEFT + WIDTH
BOTTOM = TOP + DEPTH

* read only boxed area, store away
VIDEO.RW( LEFT, TOP, RIGHT, BOTTOM, 'R', IMAGE2)

* clear boxed area and put border around it
VIDEO.RW( LEFT, TOP, RIGHT, BOTTOM, 'C', '')
BORDER.UP( LEFT, TOP, RIGHT, BOTTOM, 1, CURATR()) ; * add border

* actual processing loop
DONE = FALSE ; * flag for LOOP..REPEAT
FIRST.TIME = TRUE ; * flag indicating first time thru loop
RESP = "Y" ; * default response for message call
STATE = NULL ; * initialize scribe state to null (new)
LOOP
  IF NOT( FIRST.TIME ) THEN
    MSG("Do you wish to call SCRIBE again?","RC",RESP,'')
    IF RESP EQ "Y" THEN
      * determine whether to re-initialize the default data

```

```

        REDO = "N"
        MSG("Re-initialize IVAL (Y/N)?", "RC", REDO, "")
        IF REDO EQ "Y" THEN
            * set IVAL and STATE to null
            IVAL = NULL ; STATE = NULL
        END ; * if redo eq 'y'
        END ; * if resp eq 'y'
    END ; * if not( first.time )

```

```

UNTIL RESP NE "Y" DO

```

```

    * call SCRIBE to fit into existing window
    * if first time, load IVAL w/ default data, otherwise
    * accept existing data
    IF FIRST.TIME THEN
        * note use of @vm as delimiter (as in DELIM)
        IVAL = "This is a" :@VM: "sample call" :@VM: "to SCRIBE"
    END ; * if first.time

```

```

    PROTECT = FALSE                ; * protects IVAL if true
    EXIT KEY  = ESC:@FM:F9         ; * keys that cause exit
    EXCEPT KEY = CURSOR.DOWN     ; * causes exit w/ editor OFF
    COL = LEFT+1                   ; * left column
    ROW = TOP +1                   ; * top row
    DISPLEN = (RIGHT-LEFT-2)       ; * display length
    COLOR = NULL                   ; * video attr (null = default)
    * load values out of the environment
    COLOR<1> = @XW<CTEXT>
    COLOR<2> = @XW<CSEL>           ; * for edit window
    COLOR<3> = @XW<CCOMMENT>

```

```

    FLAGS                = NULL    ; * environment flags for SCRIBE
    FLAGS<ST.CHAR$>      = NULL
    FLAGS<NOPAINT$>      = FALSE
    FLAGS<UPPER.CASE$>   = FALSE
    FLAGS<EDIT.NAME$>    = "SCRIBE SAMPLE"
    FLAGS<FORCE.EDIT$>   = TRUE
    FLAGS<REPAINT$>      = 0
    FLAGS<NOSUBVALUES$>  = FALSE
    FLAGS<SAVE.STATE$>   = TRUE
    FLAGS<NOZOOM$>       = FALSE
    FLAGS<NO.ORIG.VAL$>  = FALSE
    CHANGE = NULL
    LINES  = (BOTTOM-TOP)-2
    EXIT   = NULL
    INPAT  = NULL
    OUTPAT = NULL
    JUST   = 'L'
    DELIM  = @VM
    MASK   = NULL
    LINELEN = DISPLEN+1

```

```

    * STATE contains information about the last time SCRIBE was
    * called

```

```

IF FIRST.TIME THEN STATE = NULL

SCRIBE (IVAL, PROTECT, EXIT KEY, EXCEPT KEY, COL, ROW, ⇨
        DISPLEN, COLOR, FLAGS, CHANGE, LINES, EXIT, INPAT, OUTPAT, ⇨
        JUST, DELIM, MASK, LINELEN, STATE)

FIRST.TIME = FALSE

* any value returned from the window?
IF IVAL THEN
    MSG("Value = ":IVAL, '', '', '')
END

IF FLAGS<SAVE.STATE$> THEN
    SS = STATE
    CONVERT @RM TO @FM IN SS
    STRING = ''
    STRING<1,1> = "Orig value = ":SS<SS.ORIG.VAL>
    STRING<1,3> = "New value = ":IVAL
    MSG(STRING, '', '', '')
    MSG("Cleanup status = ":SS<SS.STATE>, '', '', '')
END

* display values of change and key variables
STRING = NULL
STRING<1,1> = "CHANGE = ":CHANGE
STRING<1,2> = "EXIT = ":SEQ(EXIT[1,1]):" ":SEQ(EXIT[2,1])
MSG(STRING, '', '', '')

REPEAT

* clean up
VIDEO.RW (LEFT, TOP, RIGHT, BOTTOM, 'W', IMAGE2)
VIDEO.RW (0, 0, @CRTWIDE-1, @CRTHIGH-1, 'W', IMAGE1)
STOP

```

STATUP

The subroutine STATUP allows you to write information to the Advanced Revelation status line.

STATUP (operation, location, data)

Using STATUP

The Advanced Revelation status line consists of 10 cells (3 on top, 7 on the bottom), each of which has a fixed length. By convention, certain cells tend to be used for specific functions (although you may use any but cell 10):

Cell	Length	Used For
Top		
1	6 char	name of current process (e.g. Paint)
2	6 char	current function (e.g. Enter, Roam)
3	64 char	dynamic information (program-specific)
Bottom		
4	6 char	editor status
5	6 char	(varies)
6	11 char	(varies; Row indicator)
7	8 char	(varies)
8	15 char	User name
9	18 char	"Updating Indexes" indicator
10	8 char	Level indicator

☞ This describes the status line for Help level 0 (advanced). Help levels 1 and 2 reserve the bottom portion of the status line for their own use.

When certain programs run, it appears as if cell 3 comprises more than 1 cell; this is an illusion generated by embedding vertical lines (char 179) into the string placed into cell 3.

Most Advanced Revelation programs “write” to the status line in order to indicate their current status. This status line updating can take 3 forms:

- Display a pre-defined template
- Update one or more cells directly
- Display a template and update all or a portion of cell 3

Dynamically-updated information is usually placed into cell 3 (and the system provides a mechanism to do this), but other cells can be used for this purpose if desired.

operation

Determines the type of update operation to perform.

The following are the definitions for the STATUP operators (these are also defined in the STATUS.CONSTANTS record of the INCLUDE file):

```
EQU MULTI.MODAL$ TO 1
EQU SINGLE$ TO 2
EQU MULTI$ TO 3
EQU PUSH$ TO 4
EQU POP$ TO 5
EQU SINGLE.MODAL$ TO 6
```

The operations performed are described below:

Operator	Operation
PUSH\$	Stores the current status line (those cells containing data) in <i>data</i> .
POP\$	Restores and displays the status line in the data variable saved by a preceding PUSH\$ operation. Note that only status line cells containing data at the time of the PUSH\$ operation are restored.
SINGLE\$	Updates a single status line cell.
MULTI\$	Updates multiple status line cells.
SINGLE.MODAL\$	Updates a single data column of cell 3 in the status line (does not affect constituent cell 3 data).
MULTI.MODAL\$	Updates multiple data columns of cell 3 in the status line (does not affect constituent cell 3 data).

location

The location of the data to use in the update process. The value is determined by the *operation* specified:

Operation	Value
PUSH\$	A template name in the STATUS.LIST record. If null, the current status line template (in @STATLIST) is used. Note! You cannot use the PUSH\$ operation and pass a null value if your Help level is 1 or 2 — you <i>must</i> pass a template name.

(continued)

Operation	Value
POP\$	Unused (should be null).
SINGLE\$	An integer indicating the status line cell to update.
MULTI\$	A dynamic array of integers delimited by or @FM) indicating the status line cells to update.
SINGLE.MODAL\$	This must be 1 or null, indicating the starting data column in cell 3 to be updated. The data column (and any formatting) is specified in the STATUS.LIST template.
MULTI.MODAL\$	A dynamic array of integers (delimited by or @FM) indicating the starting data columns in cell 3 to be updated. The numbers correspond to the fields in <i>data</i> . The data columns (and any formatting) are specified in the STATUS.LIST template.

data

The status line display data. The value is determined by the *operation* specified:

Operation	Value
PUSH\$	A variable in which to save the current status line.
POP\$	The variable used by a preceding PUSH\$ operation to store the status line.
SINGLE\$	The data to display at the specified <i>location</i> . Data exceeding the cell length is truncated.
MULTI\$	Dynamic array of data (delimited by or @FM) to display at the specified <i>location</i> . Data exceeding the cell length is truncated.
SINGLE.MODAL\$	The data to display in the column of cell 3 of the status line referenced by <i>location</i> .
MULTI.MODAL\$	A dynamic array of data (delimited by or @FM) to display in the columns of cell 3 of the status line referenced by <i>location</i> .

Correct Use of STATUP

```

* program to call statup
*
* this program requires a new template in the STATUS.LIST
* record in the SYSTEM file. See code for details on layout of
* this item.
!
$INSERT INCLUDE,STATUS.CONSTANTS
DECLARE SUBROUTINE INPUT.CHAR, MSG
!

* Save the current status line into the variable STATUP.STATE
* and display the status line 'MYPROG' from the system file
* record 'STATUS.LIST'
*
* (this is the layout of the ``template'' MYPROG in STATUS.LIST
* in SYSTEM). Before using, remove the leading star from each
* line (leave the 2nd star in the 4th line). Once in the SYSTEM
* file, log off, then back on in order to re-initialize the
* status list. Layout starts next line.
*MYPROG
*MYPROG2Update2Updating invoice number      ,      to go      2
*25nR#4231nR#5
**
* previous line was last line of layout

STATUP (PUSH$, 'MYPROG', STATUP.STATE)

* there are 10 positions in the status line
* the following updates each cell individually by using
* the SINGLE$ function
MSG("Ready to update one cell at a time","", "", "")
FOR I = 1 TO 10
  STATUP (SINGLE$, I, 'TEST':I)
  * used as a pause
  INPUT.CHAR (KEY)
NEXT I

* pause between printing and clearing
INPUT.CHAR (KEY)

* clear the status line 1 position at a time
MSG("Ready to clear one cell at a time","", "", "")
FOR I = 1 TO 10
  STATUP (SINGLE$, I, '')
  INPUT.CHAR (KEY)
NEXT I

* final pause
INPUT.CHAR (KEY)

* restore status line
STATUP (POP$, '', STATUP.STATE)

```

```

* the following updates more than one cell concurrently using
* the MULTI$ function. Arguments 2 and 3 are FM-delimited
* arrays containing positions and data strings, respectively.

* save old state again
STATUP(PUSH$, '', STATUP.STATE)
MSG("Ready to clear all cells", "", "", "")

* clear current status line by blanking all cells
POS = '' ; STRING = ''
FOR X = 1 TO 10
    POS<X> = X ; * build array of all positions
    STRING<X> = SPACE(65) ; * build array of blanks
NEXT
STATUP(MULTI$, POS, STRING)
INPUT.CHAR(X) ; * pause

* write real data to several positions concurrently
MSG("Ready to update several cells", "", "", "")
POS = 1 : @FM : 2 : @FM : 3 : @FM : 4 : @FM : 5
STRING = "AAAAA" : @FM : "BBBBB" : @FM : STR("C", 65) : @FM
STRING := 'DDDDD' : @FM : 'EEEE'
STATUP(MULTI$, POS, STRING)
INPUT.CHAR(X)

* restore old line again
STATUP(POP$, '', STATUP.STATE)

* the following illustrates the MODAL$ capability. This
* updates a portion of cell 3 (only), while leaving the rest
* of the cell intact. It requires a format specification in
* the status line template ``record'' in the STATUS.LIST record
* in the SYSTEM file.
* save current state again
STATUP(PUSH$, "MYPROG", STATUP.STATE)
MSG("Ready to update one position in cell 3", "", "", "")

* create loop to display loop counter on cell 3
FOR I = 1 TO 10
    * displays value of X at specific location in cell 3 (as
    * determined by value 1 in format specs)
    STATUP(SINGLE.MODAL$, 1, I)
    INPUT.CHAR(X)
NEXT I

* this time, display more than one value in cell 3
MSG("Ready to update two positions in cell 3", "", "", "")
FOR I = 1 TO 10
    DISP = I : @FM : (10-I)
    STATUP(MULTI.MODAL$, 1:@FM:2, DISP)
    INPUT.CHAR(X)
NEXT I

```



```
INPUT.CHAR(X)
```

```
* restore original line again
STATUP(POP$, '', STATUP.STATE)
STOP
```

VIDEO.RW

VIDEO.RW enables you to save, restore, and clear specific areas of the CRT. This is useful in the Advanced Revelation window environment to preserve the status of the environment in which the program is running.

VIDEO.RW addresses areas of the screen by indicating the left and right column numbers (0-79), and the top and bottom row numbers (0-24). (Note: VIDEO.RW cannot write into the area reserved for the status line.) A program can use these four coordinates to create the dimensions of a box. The box then can be read, cleared, written to, and then restored to its original state.

```
VIDEO.RW( lcol, top_row, rcol, bot_row, operator, image )
```

lcol

An integer indicating the starting (left) column of the display area.

top_row

An integer indicating the top row of the display area.

rcol

An integer indicating the ending (right) column of the display area.

bot_row

An integer indicating the bottom row of the display area.

operator

One of the following operators specifying the function VIDEO.RW is to perform:

Operator	Function
R	Save (read) the specified area of the display
W	Restore (write) the specified area of the display
C	Clear the specified area of the display

image

A variable whose value is determined by the *operator* specified:

Operator	Image Value
R	A variable used by VIDEO.RW to save the display image described by the coordinates specified in the argument list. Multiple full-screen save operations should be avoided to conserve memory. Each full screen saved requires approximately 4K.
W	The display image saved in the preceding read operation. The display image is written to the coordinates specified in the argument list.
C	A two-byte sequence specifying the background character and foreground video attribute. The background character defaults to space and the foreground attribute to Normal. For a standard monochrome display image may be null. The background character is specified as a single ASCII value. The video conforms to the structure of the DOS video attribute.

Values returned

If a read or write operation is specified, VIDEO.RW passes the display image in the *image* parameter of the argument list.

Correct Use of VIDEO.RW

```

DECLARE SUBROUTINE VIDEO.RW,MSG
* save full screen image, clear using default attributes,
* display message, and restore
VIDEO.RW(0,0,@CRTWIDE-1,@CRTMAXHIGH-1,'R',IMAGE)
VIDEO.RW(0,0,@CRTWIDE-1,@CRTMAXHIGH-1,'C','')

MSG('Saved the status line as well','','','')
VIDEO.RW(0,0,@CRTWIDE-1,@CRTMAXHIGH-1,'W',IMAGE)

* save half screen image, clear, display message, and restore
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'R',IMAGE)
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'C','')
MSG('Cleared fully half','','','')

VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'W',IMAGE)
* save partial screen image, clear, display message, and
* restore
VIDEO.RW(20,10,60,20,'R',IMAGE)
VIDEO.RW(20,10,60,20,'C','')
MAP = ''
TYPE = 1
COL = 3
ROW = 4

```

```

MAP<TYPE> = 'A'
MAP<COL> = 24
MAP<ROW> = 16
MSG('Window',MAP,'','')
VIDEO.RW(20,10,60,20,'W',IMAGE)

* for color displays only: Save partial screen image, clear
* color display and set background attributes, display message,
* and restore.
BACK.CHAR = '*'           ;* background fill character
BACKGROUND = 5           ;* background video attribute
FOREGROUND = 14          ;* foreground video attribute
COLOR = (BACKGROUND * 16) + FOREGROUND ;* color formula
COLOR = CHAR(COLOR)
ATTRIBUTES = BACK.CHAR:COLOR
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'R',IMAGE)
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'C',ATTRIBUTES)
MSG('Clear half with a background color change',' ',' ','')
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'W',IMAGE)

* for color displays only: the same example as above, but using
* the current video settings in the system variable
* @ENVIRON.SET
$INSERT INCLUDE,ENVIRON.CONSTANTS

* background character
BACK.CHAR = CHAR( @ENVIRON.SET<E.BACK.CHAR> )
* background video attribute
BACKGROUND = @ENVIRON.SET<E.BG.BACK.CHAR>[4,1]
* foreground video attribute
FOREGROUND = @ENVIRON.SET<E.BG.FORE.COLOR>[4,1]
COLOR = (BACKGROUND * 16) + FOREGROUND ; * color formula
COLOR = CHAR(COLOR)
ATTRIBUTES = BACK.CHAR:COLOR
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'R',IMAGE)
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'C',ATTRIBUTES)
MSG('Clear half with a background color change',' ',' ','')

* restore
VIDEO.RW(0,10,@CRTWIDE-1,@CRTHIGH-1,'W',IMAGE)
STOP

```





Advanced Revelation

Version 2.0

Glossary

This glossary provides concise definitions of terms that you will encounter while using Advanced Revelation. Most of these terms are expanded upon in the main documentation. Please check the index, following this section, for further information.

The **bold** words in a definition denote additional entries in the glossary.





Glossary

Account	A group of related files . An account is usually associated with a single application . Security and environment settings can be applied to the files as a group by defining the settings for the account.
Active key list	A list of record keys available for a process. For example, if you execute the R/LIST SELECT command, you will generate an active key list that can be passed on to the next process, such as a RECORDCOPY command, a window , or your own program.
Anchored menu	The menu to which Advanced Revelation returns when you leave a window or when a program finishes executing. If the menu from which you choose a window or program is anchored, you will return to that menu. Otherwise, you will return to the last anchored menu.
Application	A collection of menus , windows , reports , and other processes all related to a particular task. An example is an accounting application. Typically, an application will have its own set of files (application files) and its own main menu, from which a user may access data entry windows, reports, update procedures, etc.
Application files	All files associated with an application . Application files include data files plus the files that contain window , menu , report , and popup specifications, as well as R/BASIC programs. Each application's files are generally kept in a separate account to enable the system administrator to control security and other attributes.
Application window	A window in which data is entered and saved in a file .
Argument	In R/BASIC , a value passed to a function or subroutine . For example, a function may accept two arguments, average them, and pass back a result.
Associated multivalues (AMVs)	In a window , two or more multivalued fields whose values are related, or associated. For example, in a clients window with associated multivalues for contact name and phone number, the first contact name and the first phone number are associated, as are the second contact name and phone number, and so on. Advanced Revelation's window processor carefully regulates the values in associated multivalues. For example, if you delete a contact name, its associated phone number is also deleted.

Attaching files	The process of making files available to Advanced Revelation. When you first log on, files on the default volumes are automatically attached to the Advanced Revelation system. You can access other files by attaching the volumes in which those files reside.
Batch processing (TCL commands)	Executing a sequence of TCL commands using a single command. A batch command is an entry in the VOC file that begins with the keyword TCL , and is followed by any number of TCL commands.
Binding	The process of associating prompts in a window with corresponding fields in a dictionary while using Paint .
Bonded file	A non-Advanced Revelation file in a foreign environment that is made accessible (bonded) to Advanced Revelation through an Environmental Bond . Whether bonded files are created within the foreign environment or from Advanced Revelation, they reside in the foreign environment.
Breaking (on)	Grouping
Break table	In the R/BASIC debugger , the table that lists the places and conditions chosen to temporarily halt program execution.
Browse list	A list of records available in the current window . You can move through a browse list (display another record) using the keys [Alt-F] (forward) and [Alt-B] (back).
Btree indexing	One of the indexing methods available in Advanced Revelation. Btree indexing indexes the entire contents of a field (in contrast, see Cross Reference indexing).
Buffer	A location in memory used to store data temporarily. Buffers are frequently used to keep data available in RAM so that subsequent access to the data is faster than re-reading it from disk.
Bumping	Increasing the number of network users for Advanced Revelation with a Network Bump Disk . When you first bump your system you also indicate the network that you will be using.
Cataloging	Creating an entry in the VOC (vocabulary file) for an R/BASIC program using the MAKEVOC command . Once a program has been cataloged, you can execute it from TCL (like any other command) or from a window (using a code and command).

Code and command	The means by which you specify special processing for menu choices, window prompts , or other operations. The code indicates the type of operation to be performed (W for window , M for menu , P for popup , S for R/BASIC subroutine , B for Btree index lookup, etc.), while the command indicates the specifics (usually the name of the window, menu, popup, etc.).
Collector window	A window in which data is entered for use by a process. The data is not saved in a file .
Column	The SQL term for a field .
Command window	A window that can be used in place of a TCL command. For example, there is a command window associated with the RECORDCOPY command.
Committing	The process of closing a successful transaction . Once you have committed a transaction, Advanced Revelation guarantees that all the updates you have requested during that transaction will be made successfully, even if the system crashes.
Commit log	A central record of committed transactions for all workstations on a network . If queuing (a commit queue) is in effect, transaction and associated file references are also displayed.
Commit queue	A queue (time-ordered sequence) of committed transactions. This ensures that transactions are executed in the order in which they were committed.
Compiling	In R/BASIC , converting R/BASIC source code into executable object code .
Concatenation	Joining one string of characters to another. For example, you might concatenate the city, state, and zip fields in a record to form the last line in a mailing label.
Concept Help	Information about the overall purpose of a window or other process, displayed by pressing [Ctrl-F2]. For example, if you press [Ctrl-F2] at the Definefile window, you will learn about the process of defining files .
Concurrency control	The process of managing multiuser access to records and files . Concurrency control typically includes locking and transaction processing . Concurrency control can be adjusted to different security or data integrity needs.

Consistency level	The type and duration of locking . Used only in Advanced Revelation SQL .
Constant	A string or a number that will not change value during a process (such as a program or query). Another term for a literal , as distinct from a variable .
Context Help	Information about the current prompt in a window or element of a command, displayed by pressing [F1]. Also known as Detail Help.
Control features	A set of features including transaction processing and domain validation that you can apply to a file . To do so, you must first tag a file with the CONTROL_ON command or window . Files (tables) created using the SQL CREATE TABLE statement automatically have control features applied to them.
Control files	The Advanced Revelation system files used with bonded files to maintain Advanced Revelation-specific dictionary , indexing , and volume information.
Conventional memory	The 640K of memory directly addressable by DOS. Memory above this limit is either expanded or extended memory .
Conversion	Changing data from one format to another. Data is usually converted into internal format to save space or make calculations easier. It is then converted into external format for display. For example, the string of numbers 2066439898 may be converted to display as the phone number (206) 643-9898. Advanced Revelation includes facilities to convert date, time, money, and other common data formats.
Coordinated lock	A lock type in which a file lock is set, followed by a record lock . This prevents others from locking the file as a whole, or any record in it.
Cross Reference indexing	A variant of Btree indexing in which the data in a field is subdivided into words, and each word is indexed individually. Cross Reference indexing also allows you to indicate a stop list , a list of words <i>not</i> to index.
Cursor (screen)	The blinking line or box that indicates your position on the screen.
Cursor (select list)	In R/BASIC and embedded SQL , a temporary list or table that stores the record keys or rows (SQL) assembled by a SELECT operation. A cursor maintains a position reference, which specifies a particular record or row in the cursor.

Data entry window	A synonym for an application window — a window in which data is entered and saved to a file .
Data file	A file storing “raw” data. The definitions or descriptions necessary to interpret the data are stored in the dictionary file .
Data formatting	The process of converting data into a format suitable for display. A conversion is typically done for data that is stored in internal format .
Data type	A label that identifies the type or kind of data, such as DATE, DOLLARS, INTEGER, that will be stored in a field . This information is used to validate the data entered for a field, to format the data appropriately, and, for Environmental Bonds , to map data between differing file types .
Data validation	The process of testing data at data entry to confirm that it matches a specified pattern. Advanced Revelation contains several pre-defined data validation patterns and also provides the facility for user-defined patterns.
Database server	A processor available on a shared system (such as a network) that can accept and fulfill requests for specific sets of data. In general, a database server will accept a high-level query (such as an SQL command), process it, and return only the data requested by the command (if any). In contrast, a file server does not return specific sets of data, only entire files .
Deadlock	On network or multiuser systems, a situation in which two (or more) workstations each attempt to place a lock on a file or record already locked by the other workstation.
Default	A value that is used if no other is provided. For example, at data entry or in SQL , the default is the value that is assigned to a field or column if none is entered. Advanced Revelation also provides system defaults for such items as the default menu when logging on.
Delimiters	The character or characters used to separate parts of a command, portions of a file or field name, words in a field, elements in a dynamic array , etc. Advanced Revelation typically uses spaces as delimiters in commands, periods or underscores in the names of files or fields, and various system delimiters for the data in records .
Detaching files	Removing files from the list of currently available files. Files can be detached by executing the TCL command DETACH or by logging to another account , which detaches the files in

	the current account and attaches the files in the new account.
Detail Help	Information about the current prompt in a window or element in a command, displayed by pressing [F1]. Also known as Context Help.
Dictionary	A file that contains the specifications for the fields in a data file . Each entry, or record , in a dictionary file corresponds to a field in the related data file. An entry defines field characteristics such as dictionary type , data type , justification , display length, column heading, input validations , and output format. If the field is a symbolic field , the dictionary item also contains the R/BASIC formula that defines the field.
Dictionary formula	An expression (formula) used to define a symbolic field in a dictionary . Formulas function much the way calculated cells do in a spreadsheet. The expression is processed whenever the field is referenced. Each formula can contain up to 32K of R/BASIC code.
Dictionary item	A record in the dictionary corresponding to one field in the related data file .
Dictionary type	The tag in a field definition that identifies whether a field is a Field, Symbolic , or Group . The type determines how the contents of the field are processed when the field is referenced.
Domain controls	In R/BASIC, the ability to have Advanced Revelation check and process conversions and data validations automatically. Domain controls can be used only with cursor-oriented file operations.
Domain validation	The data validation part of domain controls .
Dragging	In Paint , moving a prompt or a label to a new position on the screen.
Dynamic array	In R/BASIC, a string of characters divided into fields , values , subvalues , etc., using special system delimiters . R/BASIC treats records from a file as dynamic arrays. A dynamic array is an array because the entire string can be accessed using a single name (usually a variable name). It is dynamic because elements can be added or removed at any time. Compare matrix .
EasyWriter	A simple means of producing R/LIST reports without knowing R/LIST commands or syntax. EasyWriter presents you

with a series of **popups**, from which you select a **file**, **fields**, and formatting options for a report.

Edit mask	A character or character sequence used to represent the acceptable length and format for data to be entered. For example, a date prompt in a window may display the edit mask "MM/DD/YYYY". An edit mask does not force the data to conform to a specific pattern — that function is accomplished by a data validation pattern.
Embedded SQL	In Advanced Revelation, SQL statements included in an R/BASIC program using special syntax.
EMS	Expanded Memory Specification. An industry standard for allocating and managing memory above conventional memory .
Encryption	Encoding window templates , popups , or passwords so they cannot be modified. Encryption has no effect on the appearance or operation of the item being encrypted.
Entry line	The display line usually following a prompt label where data is to be entered. Also referred to as a prompt .
Environment	A group of settings that determine the behavior allowed to a user or within an account . For example, a user environment will determine what menu that user will see at logon, what Help level the user will use, whether the user can access TCL , etc.
Environment windows	The windows used to control the environment for users and accounts .
Environmental Bonding	The ability of Advanced Revelation to access data stored in the format of a foreign environment . Environmental Bonding enables you to treat files in dBASE III, ASCII, and other application environments as if they were Advanced Revelation files.
Exclusive lock	A lock that prevents users from setting any other type of lock on that file or record . An exclusive lock is used if the file or record will be updated .
Expanded memory	Memory above the conventional memory limit made available using bank-switching methods. Unlike extended memory , expanded memory is not directly addressable by an application.
Expanded Memory Manager	Software that allocates and manages expanded memory .

Explicit locking	File or record locking that is activated on command rather than automatically (implicit locking).
Exploding sort	Sorting records in a file according to the contents of a multivalued field . When such a sort is executed, each value in the multivalued field is treated as if it belonged in a separate record; that is, the record is “exploded”. For example, if there are three names in a multivalued CONTACTS field, an exploded sort by CONTACTS will display the record three times on a report , once for each name in the field.
Expression	In R/BASIC , a variable , literal , string , equation, system function , or any combination of these.
Extended memory	Directly addressable memory above the conventional memory (640K) limit. Extended memory is only available on 80286- and 80386-based PCs.
External format	The format in which data is to be displayed, in contrast to the format in which it is stored. For example, the internal format of a date might be “8037”. By executing a conversion, “8037” could be converted to one of many external formats, including “01/01/1990”, “JAN 01, 1990”, or “01*01*90”.
External function	In R/BASIC , a cataloged , user-defined function, which returns a single value in its place when called from another program.
External subroutine	In R/BASIC , a cataloged , user-defined program that can be called from another program to perform a specific operation. Subroutines are created, compiled , stored, and executed independently from main programs.
Field	A subdivision of a record . Typically, each field holds a discrete piece of data such as a name, an invoice amount, a date, etc. Fields in records are called columns in SQL rows .
Field attribute	One of the characteristics in the definition of a field in a data file . Field attributes include dictionary type , data type , justification , display length, data validations , output format, etc. If the field is a symbolic field , one of the field’s attributes is the R/BASIC formula that defines the field. All of the attributes for one field constitute a single record in the dictionary for a file.
Field mark	The delimiter that separates the fields in a record from one another. In R/BASIC , the field mark, ASCII character 254, is designated as @FM.

Field position	A number identifying the position of a field within a record (first, second, etc.).
File	A storage location for data. Any data that you save — whether customer information, R/BASIC programs, TCL commands, indexing information, or any other type — must be saved into a file. Advanced Revelation includes a variety of tools to define, create, maintain, and query files in your system.
File handle	In R/BASIC, a variable that holds the information used to access a file in subsequent commands. The information in the file handle is established by an R/BASIC OPEN command.
File lock	A lock that is set on an entire file at once, preventing other users from also locking the file.
File synonym	See Qfile .
File type	A designation that indicates to Advanced Revelation how data is to be stored in and retrieved from a file . Advanced Revelation offers four user-accessible file types (Linear Hash , ROS , Revelation G LINK , and DOS), each with different features and characteristics. In addition, each Environmental Bond offers an additional file type.
File server	On a network , a processor that fulfills requests for data from a central source. File servers typically do no processing of the data, in contrast to database servers .
Filter	An R/LIST SELECT statement that, when executed, generates a select list , or list of record keys . The resulting select list identifies the records to be used for subsequent access or update operations. Filters may be saved and, once saved, may be activated or deactivated at any time.
Fixed-length field	A field in which the amount of space the data will take up is pre-determined. If the data is longer than the pre-determined length, it is truncated. If the data is shorter than the fixed length, it is padded with spaces or other characters until it is long enough.
Flag	In the Merge processor, a word or phrase that functions as a command rather than as literal text. Flags are used to insert field values, control printing characteristics (font information), change paragraph formatting, etc. Flags are enclosed in chevrons (« ») or braces ({}).
Flat-file architecture	A database design in which records are stored and retrieved based on a sequential order. In contrast to a relational

	database architecture, records are identified by their position within the file .
Footling, Footer	Text that appears at the bottom of each page of a report .
Foreign environment	An application environment other than Advanced Revelation, such as dBASE III, Lotus 1-2-3, Microsoft's SQL Server, etc. Advanced Revelation can access data stored in the format of a foreign environment by using an Environmental Bond .
Foreign file	In Advanced Revelation's Environmental Bonding , foreign file refers to a file as it is accessed and maintained by the foreign environment . In contrast, if Advanced Revelation can access the foreign file via an Environmental Bond , the file is said to be a bonded file .
Foreign key	A record key in one file that is stored as the value of a field in another file (a join field). For example, a customer number (a customer record key) stored in an invoice record is a foreign key.
Form	A type of report , much like a printed form, in which the data always appears in the same place, usually in combination with literal text that identifies the data. In Advanced Revelation, form reports are laid out using Paint .
Formula	A calculation or expression that defines a symbolic field . Formulas consist of one or more lines of R/BASIC code that tell Advanced Revelation how to derive the data for that field. Formulas can be up to 32K long.
Frame	The minimal storage unit in an Advanced Revelation file . Advanced Revelation files are divided into groups , each consisting of one or more frames. Each frame holds a series of records .
Function	In R/BASIC , a command that returns a single value in its place. For example, the function SQRT(X) returns the square root of the data in the variable X .
General Help	An overview of Advanced Revelation's online help system, available by pressing [Ctrl-F1].
Global file	A file that belongs to the special account GLOBAL . Files in this account are accessible from all other accounts.
Global variable	In R/BASIC , a variable that maintains its value across programs. R/BASIC offers two types of global variables: the user variables @USERn and @RECURn , and labeled COMMON areas.

Group	A logical division within an Advanced Revelation file . Each file is divided up into groups that store one or more records .
Group (dictionary item)	A dictionary item used as a “macro” field by R/LIST , which can include any portion of an R/LIST command except the command and file name. For example, the group field LABEL may bundle together the fields NAME , ADDRESS , CITY , ST , and ZIP for use in a report . Group fields can also include sort and selection criteria.
Group Format Error (GFE)	An internal error in the structure of a Linear Hash file that makes portions of the file inaccessible. The TCL command DUMP can be used to examine files and repair GFEs .
Grouping	Creating subdivisions in an R/LIST or SQL report according to data that records have in common (also called “breaking on” a value). For example, you may group a report by the ST (state) field . Your report will then show a division (a blank line or page break) each time a new state is displayed on the report. Grouping is often used in conjunction with sorting .
Hashing	A means of calculating the location of data in a file using a record’s key , without recourse to an index . When using hashing, a filing system can go directly to a record in a file, resulting in superior performance, particularly in large files and on networks .
Heading, Header	Text that is displayed at the top of each page of a report .
Help level	The setting that determines the degree of assistance that you automatically receive while working in Advanced Revelation. The Novice help level automatically provides explanations of each process that you access. The Intermediate help level provides a list of active keys on the status line . The Advanced help level provides no special assistance. At every help level you have access to Advanced Revelation’s online General , Concept , and Context help , which are available for every operation.
Implicit locking	Automatic file and record locking . Most Advanced Revelation tools (for example, Advanced Revelation windows and the EDIT process) lock files or records as you access them. In addition, Advanced Revelation’s SQL offers a variety of options for implicit locking. In contrast, R/BASIC programmers typically implement explicit locking, setting and releasing locks under program control.
Implicit record key	In an Environmental Bond to a flat-file structure, a number that identifies the position of a record in the file.

Implicit keys are automatically (implicitly) assigned as the **record key**.

Index, Indexing	A sorted list of record keys or the process of creating and maintaining such a list. An index is sorted according to the data in the indexed field. For example, if you index the company name field in a file, the index is alphabetized by company name. Indexing has three advantages: you can access records from a data entry window using an indexed field(s) instead of record keys; selecting records for reports and other processes is faster when you select based on indexed fields; and sorting records is faster, because the index is already sorted. Advanced Revelation offers four types of indexes, each with different characteristics: Btree indexes , Cross Reference indexes , Relational indexes , and Quickdaxes/Rightdaxes .
Inline formulas	Formulas for symbolic fields that can be created directly in an R/LIST command.
Input validation	Data validation .
Interactive SQL	SQL commands that are typed in at a prompt in a window (as distinct from embedded SQL or the SQL Module Language).
Internal format	The format in which data is stored. The internal format of data is usually more compact than the external format and makes calculations or other operations easier.
Internal function	In R/BASIC , a function that is already available as part of the R/BASIC compiler (in contrast to user-defined external functions).
Internal subroutine	In R/BASIC , a module that can be called from anywhere in a program, but that is still part of the same program. Compare External subroutine .
Join field	A field in one file that stores keys to another file (a foreign key).
Joining	Bringing two or more files together in a single operation — such as a query — based on data that they have in common. For example, you might join the customer and invoice files to create a report that displays all the information required to print an invoice.
Justification	The orientation of data within a field or display column. Justification options include left (data is displayed flush left), right, and centered. For display purposes, you can also select

	text justification, which wraps long fields into a paragraph when the data is displayed.
Keystroke capture	Recording the keystrokes required to execute a process. Captured keystrokes can be played back later to repeat the process without having to repeat each keystroke.
Keyword	A word or phrase that is part of the basic vocabulary of a language such as SQL or R/BASIC .
Label	a) As a report , a report format that is used primarily to create mailing labels. b) In R/BASIC , a tag that identifies a particular location within a program. c) Prompt label .
Labeled COMMON	In R/BASIC , a set of global variables that can be accessed from any program simply by referencing the appropriate name.
Latent select list	A list of record keys that has been requested (as with an R/LIST SELECT command) but not yet assembled. A latent list is resolved (assembled) only when a subsequent process uses the list. Using a latent list can be faster than first resolving the list and then calling another process because, with a latent list, each record is only read once.
Linear Hash file	Advanced Revelation's default file type . A Linear Hash file expands and contracts as data is entered or removed from it, while retaining the same efficiency.
LINK file	One of two file types available under earlier versions of Revelation. A LINK file is efficient for retrieval but does not resize itself as does a Linear Hash file . As a result, the system administrator had to periodically resize the file to match the quantity of data in it.
List (report)	A report format in which records are shown as rows and fields are shown as columns. In Advanced Revelation you can create list reports using R/LIST and SQL .
Literal	Strictly defined, a value that does not change. In R/LIST and SQL , this typically means a value to compare a field against. In R/BASIC , a literal is an alphanumeric value assigned to a variable or used in an expression . Literals must always appear in quotes, single or double. See also constant .
Loading	Calling a process or template into memory to be executed. For example, an R/BASIC program must be loaded into RAM from disk before it will run.

Local Area Network (LAN)	A group of PCs linked together so they can share resources such as a printer or disk, or a database of information.
Local subroutine	See Internal subroutine .
Lock, Locking	The process of flagging a file or record as “in use”. While a file or record is locked, no other user can lock it, and no other user should attempt to update it. The file or record is still available, however, so that other users can use it in reports or other non-update operations.
Lock waiting	On network or multiuser systems, a situation in which a workstation is waiting to attempt to place an exclusive lock on a file or record , but is continually defeated by other workstations that are placing shared locks . Advanced Revelation includes controls that allow you to specify how long a workstation will wait before canceling its effort to lock the file or record.
Logon option	A command, word, or switch you can add while logging into Advanced Revelation from the operating system . These options allow you to change the way Advanced Revelation will behave. Logon options include user name, a switch to disable the math coprocessor, switches for using expanded memory , etc.
Macro key	A user-defined keystroke that is active at any time in Advanced Revelation. You can assign any function or process to a macro key, and that process is then available throughout your system.
Maintenance release	A set of disks that update your version of Advanced Revelation. Maintenance releases only correct small problems in the software and do not usually add any new functions.
Master field	The primary definition of a field and, with Environmental Bonds , the definition that is used as the basis of all interaction with a foreign environment . All synonym fields use the master field as their basis.
Matrix	An array of variables that are accessible using a single variable name and the appropriate subscripts (for example, ARRAY(4,1). Unlike dynamic arrays , matrices must be pre-defined (dimensioned) to a fixed size.
Menu	A series of options displayed on the screen. Typically, each choice executes a different process in your system or application.

- Merge** The process of combining a **template** for a letter or other document with **records** from a database, and inserting data from each record into the document. An example is a form letter containing fixed text interspersed with places for data from a **file** (such as a customer's name, address, and balance due). When the merge process is run, the merge template is combined with the data from the file to create a personalized document, one for each record in the file. Merge differs from **Form** in that it can format varying quantities of text and data to create varying length output, while Form creates fixed-format output.
- MFS** A Modifying Filing System. A program that can monitor normal file activity such as **reading** and **writing** data to a file and interrupt the activity to do specialized processing. MFSs are used for such functions as securing data, compressing data, and maintaining audit trails.
- Modulo** a) In an Advanced Revelation **file**, the number of **groups** available for storing data. In **Linear Hash files**, the modulo can change as data is added or removed; in **LINK** and **ROS** files, the modulo is fixed at the time the file is created. b) In arithmetic operations, the modulo is the remainder in a division operation.
- Multipart key** A **record key** that is constructed from data in more than one **field**.
- Multitasking** Able to execute more than one task at the same time. For example, OS/2 is a multitasking **operating system** because, with OS/2, a user can search a database, print a document, and calculate a spreadsheet all at the same time.
- Multiuser** A computer that can be used by a number of separate users simultaneously. A typical layout is to have one central computer with several terminals (keyboard and monitor) attached. **Networks** are sometimes referred to loosely as multiuser systems because multiple users are sharing central resources.
- Multivalued field** A **field** containing more than one data element. For example, a multivalued field **CONTACTS** may contain any number of names. In Advanced Revelation files, the only limitation on the number of values in a field is that the total length of the **record** cannot exceed 64K.
- Nesting** The ability of a process to contain or call another process. For example, an R/BASIC **subroutine** can call another subroutine, thereby nesting that subroutine.

Network	Local Area Network.
Network Bump Disk	A product from Revelation Technologies that turns a single-user version of Advanced Revelation into one that can support access by up to four concurrent users on a network . If used with a system that has already been bumped, a Network Bump Disk adds four users to that copy of Advanced Revelation.
Network drivers	Modules that communicate between Advanced Revelation and a network operating system . There is a network driver for every network operating system that is supported by Advanced Revelation.
Nonconventional memory	Memory in a PC between 640K and 768K. Although reserved by the PC for specialized functions (such as video), Advanced Revelation can take advantage of this memory if it is not otherwise being used.
Normalization	The process of creating flat, single-valued records out of records that contain multiple values. For example, if a customer record contains five invoice numbers, normalizing it will create five records with one invoice number each and other data repeated as necessary.
Null	Containing no value. A null is distinct from a zero or blank value. For example, a character field that has never been assigned a value is null; if it contains blanks, it may be empty, but it is not null.
Null handling	A set of rules for how nulls in one environment (example: dBASE III) are to be translated into another environment (such as Advanced Revelation). A part of the definition of each Environmental Bond is a description for how that bond will handle nulls.
Object code	In R/BASIC, an executable program. The instructions in a program are written in English-like source code and then translated by a compiler into executable object code.
Online help	Help text and documentation that is displayed on the computer screen. In Advanced Revelation, online help is available in any window by pressing [F1] (Context Help), [Ctrl-F2] (Concept Help), or [Ctrl-F1] (General Help).
Operating system	The software that controls all interaction between the user and the hardware. The operating system is the software that handles such fundamental tasks as displaying characters on the screen or accessing a disk drive. Application software, such as Advanced Revelation, typically uses the

operating system software to accomplish these tasks. The two most popular operating systems for the PC are MS-DOS and OS/2, both published by Microsoft Corporation. In addition, many networks have their own operating system, such as Novell's NetWare or Banyan's VINES.

Operating system file	In Advanced Revelation, "operating system file" refers to any file that you access using the name assigned to it by the operating system . (You can see operating system file names using the DIR command.) In contrast, an Advanced Revelation file has a name that is exclusive to the Advanced Revelation environment, and can be accessed using any Advanced Revelation process.
Operator	The person currently using Advanced Revelation.
Optional modules	Utilities or systems that are not a part of the basic Advanced Revelation package, but can be added on. An example of an optional module is an Environmental Bond .
Orphan	In the Merge processor, a line that appears by itself at the top of a page, separated from the rest of its paragraph on the previous page. Compare Widow .
Paint	Advanced Revelation's tool for creating windows . In addition to creating basic windows, you can use Paint for customization, including video attributes , security, binding to data dictionaries, hooks for custom processes, and much more.
Panning	The process of moving through a window whose total size extends beyond its borders on the screen. The area beyond the screen borders is known as virtual space .
Parameter	A value that is passed to a process. For example, the ATTACH command at TCL accepts a parameter for the volume name, and optionally, an additional parameter that contains a list of files.
Parse, parsing	a) Processing a command or statement to separate its constituent elements. For example, in Advanced Revelation the TCL processor parses each command you enter at the TCL window into the command itself, a file name, field names, record keys, etc. b) In R/BASIC, to copy the elements of a dynamic array into a matrix .
Password	A word or set of characters that you use to secure an account or a window . If a password has been assigned to an account or window, you must enter that password correctly before Advanced Revelation will allow you to access the account or use the window.

Path	The operating system location for a file . A path can include a drive designation and the subdirectory that describes where a file resides.
Popup	A window that pops up to present you with a list of options. When you choose an option from a popup, your choice is typically returned to the place from which you called the popup. For example, a popup called from a window prompt will typically return a value to the prompt.
Popup mode	The popup mode determines whether the data displayed in a popup comes from a series of records , fields , or values within a field. You specify the popup mode when you design a popup.
Primary key	A record key , called primary because it is the primary means of record access. In contrast, indexed fields are sometimes referred to as secondary keys.
Product	In SQL , the process of multiplying two tables so that there is an entry for each combination of rows in the tables.
Program stack	The list of R/BASIC programs that are currently loaded into memory.
Projection	In SQL , the process of selecting only certain columns out of a row . For example, the command SELECT NAME, AGE FROM EMPLOYEES uses a projection specification to extract the columns NAME and AGE for rows in the EMPLOYEES table.
Prompt	An area in a window where you can enter or display data. A prompt very often has a prompt label next to it that briefly describes the purpose of the prompt. Also known as an entry line .
Prompt label	In a window , the text next to a prompt that tells you what the prompt is for.
Prototype	A preliminary model used to test a design (also used as a verb meaning “to model”). Because Advanced Revelation applications can be modified so readily, it is easy to convert a prototype into a final application, rather than starting again from scratch.
Qfile	An alternative name for a file . Any Advanced Revelation function that you can perform using a file’s original name can be performed using a Qfile name, with the exception of deleting the file. The chief benefit of Qfiles is the ability to access

files across **accounts** instead of having to maintain copies of a file in several accounts.

Qualify In **SQL**, the process of identifying a column or table name uniquely. For example, in a statement that joins two or more tables, each column name must be qualified with its table name.

Query The process of extracting data from a database, such as for a **report**. Also used to mean the actual command or process used for the extraction.

Query mode A mode for a data entry **window** that allows you to search, **sort**, and **report** data using the **prompts** in the window.

Queue A list of keys or elements. Most often used to indicate a list that has been assembled in anticipation of using it in a subsequent process.

Quick Paint In **Paint**, the process of selecting **fields** from an existing **file** for display in the **window**. This is in contrast to the more common method of creating fields while using **Paint**.

Quickdex A type of **index** that maintains an ordered list of **record keys**. In contrast, a **Btree** or **Cross Reference index** maintains an ordered list of data from a record. A **Rightdex** is like a **Quickdex**, but is used for numeric keys.

R/BASIC Advanced Revelation's programming language. **R/BASIC** is a complete programming language, comparable with languages such as **C**, but is specifically designed to be used in the Advanced Revelation environment.

R/LIST Advanced Revelation's English-like query language. **R/LIST** includes commands and options to search **files**, **sort** data, and create columnar **reports**.

RAM Random Access Memory. **RAM** is a computer's workspace, where all processing takes place. **RAM** is volatile, and its contents disappear when the power is turned off or the computer is reset.

Range variable In **SQL**, a temporary synonym for a **table** name. Range variables are used to reduce the clutter and effort of repeating the table name each time a **column** from that table appears in a command.

Read In **R/BASIC**, to fetch a **record** from a **file**.

Record	A single entry in a file , such as an entry for a single customer, invoice, or product. A record consists of fields , and a file consists of records, each with a unique name (key).
Record key	The name for a record in a file and the means by which a record is retrieved from a file. In Advanced Revelation, each record must have a unique key. A key can consist of a single field (the key field), or can be constructed out of multiple fields (a multipart key), the combination of which must be unique. To retrieve a record from a file, you must know (or be able to find) its key.
Record lock	A lock that is set on an individual record within a file , as distinct from the file as a whole.
Record mark	A special character (ASCII character 255) used to separate records in a file . In R/BASIC, a record mark is available as the variable @RM.
Recursive execution level	In Advanced Revelation, a process can be suspended and a new level of TCL opened up. Each such level is termed a "recursive execution level". A recursive level is independent of other levels, and processes executed there do not affect processes at other levels.
Register	A temporary storage area associated with a window or a prompt in a window. Registers can be accessed from within R/BASIC programs.
Relational database	A model for database architecture in which all information can be represented as rows in a table, with columns to represent the different elements of each row. New tables can be created (not as physical tables, but as representations) by joining data from various other tables.
Relational indexing	A type of index in which the index information is stored in another data file . Relational indexing is used frequently to establish a dependency between files that share information, such as between invoice and customer files.
Report library	In EasyWriter , a file containing previously-defined report specifications.
Report	An orderly display of information on the screen or printer. Creating a report typically involves specifying selection criteria to indicate which records are to be displayed, sort criteria to order the records, and the format of the report (columns, mailing labels, etc.). Advanced Revelation offers a wide variety of tools and utilities for creating reports.

Resolved select list	A list of record keys that has been completely assembled (resolved) for use by a subsequent process. The list can be used immediately, or saved for future use. Compare Latent select list .
Results table	In SQL , a pseudo-table that contains all of the rows returned by a SELECT statement. The table can be used by a subsequent process, or saved for future use. (A results table is the SQL name for a select list .)
Retrieval language	A language (command set) that is used primarily for producing reports . Advanced Revelation features the retrieval languages R/LIST and SQL .
Rightdex	A version of Quickdex in which keys are maintained in right-justified order (used with numeric keys).
Rollback	The process of canceling a transaction . If you roll back the transaction, none of the updates you have made since the transaction began will be finished, and the files affected by the transaction will remain as they were before.
Rollout file	In the DOS version of Advanced Revelation, a file into which the Advanced Revelation environment is written temporarily after you have entered the SUSPEND command at TCL . With Advanced Revelation rolled out to a file, you have more memory available to execute other procedures. When you re-enter Advanced Revelation by typing EXIT at the operating system prompt, the Advanced Revelation environment is retrieved from the rollout file and restored.
ROS file	An Advanced Revelation file type that maintains large pieces of the file in RAM (in a buffer). ROS files are useful for smaller files that are accessed frequently, but they cannot be used for shared data on a network .
Row	SQL terminology for a record , or a single entry in a table .
Runtime	a) An inexpensive version of Advanced Revelation that contains everything required to run an application but not develop or modify one. Runtime and development versions of Advanced Revelation are sold as separate products. b) In the phrase "at runtime", refers to events that occur when a process is being executed. For example, the R/BASIC language in Advanced Revelation does not require explicit data typing, but instead converts strings to numbers at runtime.
Script	In the Merge processor, the combination of text and control information that is used as the basis for a document. The

	script contains literal text plus flags that indicate the format and the data that is to be merged with the text.	
Scrolled cursor	In SQL , a cursor in which you can move forward, backward, or to a specific location.	
Selecting, Selection	Searching a file to find records meeting certain criteria. For example, if you look for all customers with a status of "active", you are selecting records out of your customer file.	
Select list	A list of record keys that is available to a subsequent process. You will typically generate a select list using an R/LIST or SQL SELECT statement, and use the resulting list in a report , program, or other process. Select lists can be saved in a file for future use.	
Sequential counter	In a window , a default value for a prompt that provides the next available number. A sequential counter is used frequently to assign the next available number to an invoice or a new customer.	
Serializable transactions	An advanced feature of transaction processing that guarantees that complex, multifile transactions are processed in the order that they are committed , thus increasing the integrity of the data files . Without this feature, user A might commit a transaction before user B, but user B may update the file before user A has had an opportunity to do so.	
Shared lock	A lock that prevents other users from setting an exclusive lock but that allows additional shared locks. Shared locks are used to indicate that a file or record is in use (as in a report) but not being updated .	
Softkey	A keystroke that can have a different definition in each window . In Advanced Revelation, each window can have up to 20 softkeys defined, each of which can execute a separate process. Pressing [F6] displays a list of softkeys defined for the current window, if any.	
Sort	To put data into a specific order, usually according to the contents of a field in a file . Although data in Advanced Revelation files is not stored in any particular order, the use of an index or of a sort option in a report can return data in almost any order required.	
Source code	In R/BASIC , a program as entered by a user. A programmer enters English-like commands such as PRINT and READ . The source code is then compiled into object code , which is the form of a program that Advanced Revelation understands and executes.	

- SQL** Structured Query Language. A language that has become standard for databases in all types of computers and environments. SQL allows you to build, maintain, and **query** databases using relatively simple commands.
- SQL Module Language** A version of **SQL** that can be **compiled** much like a programming language. The SQL Module Language is used by programmers as a high-level language that can be ported to other environments easily.
- Station** **Workstation.**
- Status line** The area at the bottom of the screen that displays information about the current process.
- Step size** The number of rows or columns that a **window** will be moved, **panned**, or resized per keystroke. If you increase the step size, each keystroke will be worth more rows or columns, making it faster for you to move or resize the window.
- String** Any sequence of characters, including alphabetic characters, numbers, punctuation, and special characters such as **delimiters**.
- String space** The **RAM** available for your applications while you are in Advanced Revelation. An "out of string space" error indicates that there is insufficient memory available for the operation you have requested.
- Stop list** A list of words associated with a **Cross Reference index** that are *not* to be indexed. A stop list usually includes words that appear so often (the, a, this, every, etc.) that they will likely not form the basis of a **query**.
- Subquery** In **SQL**, a search command embedded within another command. A subquery returns a **results table** that is the basis for further processing in the same command. Using subqueries, you can pass the results of a **query** against one table into a query against another.
- Subroutine** In **R/BASIC**, a module that can be called from anywhere in a program to perform special processing.
- Subscript** A designation indicating the location of an element in a list or table. For example, the **R/BASIC expression** @RECORD<3> uses the subscript "3" to return the third **field** from the current **record**.
- Subtext mark** A special character (ASCII character 250) sometimes used to separate elements of the same text line.

Subvalue	One of a series of elements within a single multivalued field . For example, if a file contained the two multivalued fields CONTACT and CONTACT.PHONE, the CONTACT.PHONE field might contain subvalues (multiple multivalues) corresponding to several phone numbers for each contact.
Subvalue mark	A special character (ASCII character 252) used to separate values within a multivalued field . In R/BASIC , a subvalue mark is available as the variable @SVM.
Suspend	To temporarily halt a process in order to execute another in such a way that the first process can be resumed. Advanced Revelation can suspend most processes and open a new level of TCL (a recursive execution level) to execute other Advanced Revelation processes. Advanced Revelation can also suspend itself in order to execute other operating system processes such as word processors or spreadsheets.
Symbol table	A record that contains information about all the variables used in an R/BASIC program, used primarily by Advanced Revelation's debugger.
Symbolic field	A field for which the data is not extracted directly out of a record but instead is calculated according to a formula . Examples include fields that calculate a total based on information in an invoice record and fields that look up data in a related file . The formula that defines a symbolic field can be up to 32K long. A symbolic field is normally indistinguishable from other fields in a file, and you can search, sort , and display the data represented by that field.
Synonym field	A second (third, fourth, ...) entry in a dictionary for a field that already exists in a file . Each synonym can treat the data in the field differently — for example, a synonym field may have a different display length, justification , or external format than the master field .
SYSPROG	The System Programmer account . SYSPROG is the account used to create other accounts in the system and is also the account that is used by system administrators for modifying features such as users and environments .
System administrator	The person responsible for maintaining the Advanced Revelation environment as a whole. A <i>system administrator</i> might be responsible for such duties as creating and securing new users , creating new accounts , applying updates and upgrades , backing up the system, and monitoring transaction and network functions in the system. A sys-

	tem administrator must have a comprehensive grasp of how Advanced Revelation functions.
System delimiter	Any of seven special characters (ASCII values 249 to 255) to which Advanced Revelation assigns special status. System delimiters separate records in a file (record mark) , fields within a record (field mark), values within a field (value mark), etc. Many R/BASIC commands are expressly designed to recognize system delimiters.
System files	Files shipped with Advanced Revelation that contain such system information as the basic Advanced Revelation vocabulary (VOC), the programs used by Advanced Revelation commands (VERBS and SYSOBJ), templates for system windows (COMMANDS) , and so forth. You would not normally modify these files, although there is no restriction on doing so.
System variable	In R/BASIC , a variable (such as @RECORD) that is used and maintained by Advanced Revelation. Some system variables are available for general use, while others are simply there to allow a programmer to query the status of the system.
Table	The SQL term for a file .
TCL	The Command Level. The program in Advanced Revelation that accepts interactive commands and processes them. TCL appears as a window at which you can enter any TCL, R/LIST , or SQL command.
TCL level	When you suspend a process by pressing the [F5] key, a TCL window appears. This window is one recursive execution level or TCL level higher than the process that was suspended. When you exit the TCL window, you return to the process you suspended earlier.
Template	A collection of information that describes how a window , popup , menu , report , or other function appears and operates. You can change the behavior and appearance of a window (or popup, etc.) by changing the template. The next time you run the window, it will be displayed and processed using the changed information in the template.
Terminal emulation	The ability of a PC to behave as if it were a dumb terminal connected to a mainframe or minicomputer.
Terminating/ non-terminating cursor	In R/BASIC , a select list . If the list is terminating, the record keys are removed as they are read from the list. In a non-terminating list, the keys are left in the list, and the list

	wraps around to the starting point when the last key has been read.	
Text field	A field that contains data that has been subdivided into lines using text marks .	
Text mark	A special character (ASCII character 251) used to separate lines of text. In R/BASIC , a text mark is available as the variable @TM .	
Throwaway words	In R/LIST , words that can be included in a command for readability but that are ignored by Advanced Revelation.	
Trace table	In the R/BASIC debugger, a list of lines and variables that are being tracked while you execute a program.	
Transaction	A group of updates to one or more files that are to be treated as a single unit. If designated as a transaction, either all updates will be executed or none will. This guarantees the integrity of files in which data depends on the data in other files.	
Transaction control	A feature that allows you to use files in transaction processing . You must explicitly add transaction control to a file before using it in a transaction.	
Transaction processing	The process of starting and committing a series of file updates as a unit. Transaction processing involves adding transaction control to a file, starting a transaction , updating files, and then committing a transaction.	
Type mapping	Matching a generic Advanced Revelation data type (such as INTEGER , DATE , or CHAR) to a specific data type in a foreign environment (such as the dBASE data type MEMO). A type map tells Advanced Revelation how to interpret the data type in a foreign environment and how to convert data from Advanced Revelation into the format used by that environment.	
Union	In SQL , the process of combining two tables so that the end result is a table with all rows (including duplicates) from both tables.	
Update, Updating	a) Changing the data in a file by editing it or making a new entry. b) A Maintenance release .	
Upgrade	A set of disks that modifies your version of Advanced Revelation by adding features. An Upgrade changes the version number of Advanced Revelation (such as from 2.0 to 2.1).	

Usable high RAM	Areas of RAM between 640K and 1MB that are not being used for their reserved purpose and can thus be used by Advanced Revelation. For example, an area of high RAM not being used in your computer for monochrome video can be used by Advanced Revelation in conjunction with expanded memory .
User	a) A name by which you can log into Advanced Revelation. Each user defined in Advanced Revelation can have unique environment and video attributes , security privileges, and a unique logon process. b) In a network version of Advanced Revelation (one that has been bumped), a person currently logged into the product. A copy of Advanced Revelation that has been bumped once will allow up to four users to access it simultaneously.
Validation pattern	A rule or set of rules that determines whether data entered into a field is valid. For example, a validation pattern may require that the data in a field consist of up to 10 alphabetic characters, or of three numerics, a dash, and four more numerics (a phone number).
Value	a) A single data element in a multivalued field . b) In SQL , data that is to be inserted into a row .
Value mark	A special character (ASCII character 253) used to separate values in a multivalued field . In R/BASIC , a value mark is available as the variable @VM.
Variable-length field	A field in which any quantity of data can be stored and in which the field stores only as much data as is put into it (no padding). In Advanced Revelation files (Linear Hash and ROS), all fields are variable length. Environments accessed through Environmental Bonds may not support variable-length fields.
Variables	In R/BASIC , storage locations for data that might change during the course of the program. In contrast, a literal does not change value during a program.
Verifile	The process of confirming the validity of data entered at a prompt by looking it up in another file . For example, you might check the validity of a product code in an invoice by looking it up in a product file.
Video attribute	A code or series of characters that identify the color and special formatting (blink, underline, bold) of a window , prompt , popup , menu , or other element of Advanced Revelation.

- View** In **SQL**, a pseudo-**table** that is defined as a selection of **columns** and **rows** from one or more tables. A view can be treated like a real table, but it will only display the columns and rows that are part of its definition, and is thus useful for security.
- View window** A **window** used to display **R/LIST** and **SQL reports**. The View window allows you to scroll forward and backward in the report, and side-to-side if the report is wider than the screen.
- Virtual space** The area of a **window** outside of the visible display area. The display area of a window is limited to the size of the screen. However, you can define a window as large as 160 columns by 160 rows; the area not currently visible on the screen is said to be in virtual space.
- VOC file** Advanced Revelation's vocabulary **file**. The VOC file contains an entry for every command that can be executed from **TCL**. Each **account** has its own VOC file, which can be customized by adding, renaming, or removing commands.
- Volume** A designation that identifies a combination of file location and **file type**. For example, you might define a volume called **ASCII_DATA** that tells Advanced Revelation that the files associated with that volume are ASCII files (accessed through an **Environmental Bond**) and are located in the subdirectory **C:\AREV\ASCII**.
- Volume directory** A map or table of contents that tells Advanced Revelation which **files** are associated with a particular **volume**. There is a volume directory for each combination of location and **file type**, or **volume**.
- Volume label** An internal name for a **volume directory**, which is used by several processes in Advanced Revelation (such as **indexing**) to guarantee the uniqueness of **files**.
- Widow** In the **Merge** processor, a line that appears by itself at the bottom of a page, separated from the rest of its paragraph on the next page. Compare **Orphan**.
- Window** A screen form that allows you to enter, modify, or report data. A window usually contains **prompts**, areas on the screen where you enter and display the data associated with a particular **field**.
- Window level** A number that tells you how many **windows** you currently have displayed or **suspended**. If you call a window from

another window (for example, if you use a **softkey** to display a related window), the window level goes up by one.

Window processing

The flow of control through a **window**. Advanced Revelation steps through a window according to a well-defined sequence of steps. At most steps you can interrupt window processing and add your own processing (such as a custom routine), usually by indicating a **code and command**. This gives you a very high degree of control over how the window behaves.

Word/line wrap

A feature in **text fields** or in the editor that causes the **cursor** to jump to the next line automatically when you reach the end of a line.

Workstation

A single PC on a **network**. A network usually consists of at least one central **file server**, the central resource for storage and printing, and one or more workstations.

Write

In **R/BASIC**, to save a **record** in a **file**.

Zoom

Expanding a **window** or line into a larger entry area. For instance, if you press the [F3] key while at a window **prompt**, the prompt will zoom into a larger window.



Index





; (multiple statement operator)
 (R/BASIC) N4.2
 \$INSERT (R/BASIC) N4.7 - 8
 , !, REM, / ... */ (comments)
 (R/BASIC) N4.1 - 2
 +++, ---, ***, /// (multivalued arithmetic
 operators) (R/BASIC) N4.9 - 10
 :: (multivalued concatenation)
 (R/BASIC) N4.9 - 10
 < > dynamic array extraction and
 assignment) (R/BASIC) N4.10 - 11
 =, +=, -=, := (assigning values)
 (R/BASIC) N4.11 - 15
 @ (cursor position on screen)
 (R/BASIC) N4.16 - 17
 @ variables
 See R/BASIC, variables
 [] (string extraction and assignment)
 (R/BASIC) N3.11, 4.3 - 6
 { } (accessing fields) (R/BASIC) N4.6 - 7

A

ABORT (R/BASIC) N4.17
 ABS (R/BASIC) N4.18
 Absolute value (R/BASIC) N4.18
 Accounts
 See also Chapter F3
 See also Environmental bonding
 See also Environments
 accessing via LOGTO (TCL) I3.43 - 44
 creating F3.9
 creating (TCL) I3.45 - 46
 definition F3.2 - 3
 deleting F3.9
 deleting (TCL) I3.17 - 18
 displaying current status
 (TCL) I3.89 - 90
 security F3.10 - 11
 storage requirements F3.9
 SYSPROG (System
 Programmer) F3.4, 3.9
 upgrading after software
 installation B2.6 - 10
 Active key list
 See R/BASIC, cursors
 See Record keys
 ADD (SQL) K8.10

ALL (SQL) K8.10
 ALPHA (R/BASIC) N4.19
 Alphanumeric
 See R/BASIC, strings
 ALTER TABLE (SQL) K8.31 - 33
 AMVs (associated multivalues) F5.2
 AND (R/BASIC) N4.20 - 21
 ANY (SQL) K8.11
 Application design
 See Chapter F2
 Application windows
 See Paint
 See Windows
 Arctangent N4.21 - 22
 Arguments
 See R/BASIC, subroutines and
 functions
 See R/BASIC, variables
 Arithmetic
 See R/BASIC, numbers
 Arrays
 See R/BASIC, dynamic arrays
 See R/BASIC, matrices
 AS (SQL) K8.11
 ASCII
 See also Environmental bonding,
 ASCII bond
 See also R/BASIC, formatting
 See also R/BASIC, strings
 displaying character codes D4.13
 Assignments
 See R/BASIC, variables
 Associated multivalues F5.2
 at variables
 See R/BASIC, variables
 ATAN (R/BASIC) N4.21 - 22
 ATTACH (TCL) I3.1 - 2
 AVERAGE (R/LIST) J2.7
 Average (SQL) K8.11
 AVG (SQL) K8.11
 AW (TCL) I3.2

B

Batch processing
 See TCL
 Batch updates (TCL) I3.6
 See also Chapter F10

BEGIN CASE (R/BASIC) N4.27 - 28
BEGIN TRANSACTION (SQL) K8.33
BETWEEN (SQL) K8.11
Binary values (R/BASIC)
 bit-wise operators N4.22 - 23
 conversions N4.90 - 91, 4.140 - 141
Binding methods F4.8 - 9, 5.14 - 15
Bit-wise operators (R/BASIC) N4.22 - 23
BITAND, BITOR, BITXOR, BITNOT
 (R/BASIC) N4.22 - 23
BLIST (TCL) I3.4 - 6
Blocking text
 See Editor
Bonding
 See Environmental bonding
Boolean
 See R/BASIC, formatting
 See R/BASIC, logical expressions
BORDER.UP (R/BASIC) N5.5
Borders
 See R/BASIC, system subroutines
Branching and flow
 See R/BASIC, branching and flow
 See R/BASIC, logical expressions
 See R/BASIC, variables
BREAK KEY (R/BASIC) N4.24
BREAK-ON (R/LIST) J2.7 - 9
Break table (R/BASIC) N4.46 - 47
Browse lists
 See also Query
 See also R/BASIC, cursors
 See also Record keys
 See also Select lists
 converting select lists to G1.6 - 7
 creating with indexes G3.2 - 5
 creating with the Filter key
 [Ctrl-F10] G2.1 - 5
 definition G1.4
 editing G1.5 - 6
 keystrokes (table) G1.6
 list of records changed in current
 session G1.5
 printing G1.5
 using I1.5
BTREE.EXTRACT (R/BASIC) N5.6
Btree indexes F7.3 - 4

Buffers
 See FLUSH (TCL)
 See R/BASIC, memory management
BUILD (TCL) I3.6
BUMP (TCL) I3.7
Bumping Advanced Revelation L1.4
BY (R/LIST) J2.9 - 11

C

Caches
 See Chapter B5
 See R/BASIC, memory management
CALCULATE (R/BASIC) N4.24 - 25
Calculations
 See R/BASIC, numbers
CALL (R/BASIC) N4.26 - 27
Calling subroutines and functions
 See R/BASIC, branching and flow
 See R/BASIC, subroutines and
 functions
 See R/BASIC, system subroutines
CASE (R/BASIC) N4.27 - 28
Case sensitivity
 R/BASIC N3.1
 R/LIST J1.1
CATALOG (TCL)
 See MAKEVOC (TCL)
Cataloging R/BASIC programs I3.49 - 50
CATALYST (R/BASIC) N5.7 - 10
CHAIN (R/BASIC) N4.29
Chaining programs
 See R/BASIC, subroutines and
 functions
CHAR (R/BASIC) N4.30 - 31
Character strings
 See R/BASIC, constants
 See R/BASIC, data entry and output
 See R/BASIC, strings
CLEAR (TCL)
 See CLEARLIST (TCL)
CLEAR (R/BASIC) N4.31
CLEAR COMMON (R/BASIC) N4.32
CLEAR-FILE (TCL)
 See CLEARFILE (TCL)
CLEARDATA (R/BASIC) N4.32 - 33

CLEARFILE (R/BASIC) N4.33 - 34
 CLEARFILE (TCL) I3.7 - 8
 CLEARLIST (TCL) I3.8
 CLEARSELECT (R/BASIC) N4.34 - 35
 CLOSE (SQL) K8.33
 See also Cursors (SQL)
 Codes and commands
 See also Chapter F9
 See also R/BASIC, system
 subroutines
 displaying popups F6.13 - 14
 glossary F9.5 - 12
 in index searches G3.3
 in keystroke capture F9.5
 in windows F9.1 - 5
 options F9.12 - 13
 COL-HDR-SUPP (R/LIST) J2.4
 COL1() (R/BASIC) N4.35 - 36
 COL2() (R/BASIC) N4.36 - 37
 COLHEAD (R/LIST) J2.10
 COLHEADING (R/BASIC) N4.37 - 38
 COLLECT.IXVALS
 (R/BASIC) N5.10 - 11
 Collector windows
 See Paint
 See Windows
 COLLENGTH (R/BASIC) N4.38 - 39
 Column headings
 See R/BASIC, reports
 See Reports, R/LIST
 Columns (SQL)
 See also Fields
 See also Key columns (SQL)
 adding K3.7, 8.10, 8.31 - 33
 altering table definitions K8.19,
 8.31 - 33
 averaging K4.3
 calculations K4.4
 counting rows K4.3
 date and time default values K8.13
 default values K8.14
 definition K2.2
 deleting K3.7, 8.15, 8.31 - 33
 documenting as foreign keys K8.22
 inserting values K8.17, 8.25
 maximum and minimum values K4.4
 (cont.)

Columns (SQL) (cont.)
 naming K2.5
 null values K8.20
 redefining K3.7
 requiring unique values K8.24
 selecting all K4.2
 selecting maximum and minimum
 values K8.18 - 19
 summing K4.3
 totaling K8.23
 updating values K5.4 - 6, 8.23,
 8.57 - 60
 user name default K8.25
 Columns, multivalued (SQL)
 See also Fields
 See also Key columns (SQL)
 creating K8.22
 defining K3.6
 definition K2.9 - 10
 key columns K8.17 - 18
 Command Level, The
 See TCL
 Command windows I1.5
 Comments (R/BASIC) N4.1 - 2
 COMMIT (SQL) K8.34
 Commit log
 See SQL, transaction processing
 See Transaction processing
 COMMITLOG (TCL) I3.9
 Common variables
 See R/BASIC, subroutines and
 functions
 See R/BASIC, variables
 COMMON (R/BASIC) N4.39 - 40
 Communication
 settings (TCL) I3.55
 terminal emulation (TCL) I3.84 - 85
 Comparisons
 See R/BASIC, branching and flow
 See R/BASIC, compiler
 See R/BASIC, logical expressions
 See R/BASIC, strings
 COMPILE (TCL) I3.9 - 11
 Compiling R/BASIC programs
 See R/BASIC, compiler

- Concatenation
 - strings (R/BASIC) N3.11, 4.9 - 10
 - fields (type G, Group) E1.7 - 8, 2.11 - 12, 3.6
 - Concept help
 - See Help
 - Concurrency controls (SQL) K8.55 - 56
 - See also File and record locking
 - See also SQL, transaction processing
 - Conditional branching
 - See R/BASIC, branching and flow
 - CONSISTENCY LEVEL (SQL) K8.12
 - Consistency levels (SQL) K6.5 - 6, 8.12, 8.55 - 56
 - Constants
 - See R/BASIC, constants
 - See R/BASIC, numbers
 - See R/BASIC, strings
 - Context help
 - See Help
 - CONTINUE (SQL) K8.13
 - Control files
 - See Environmental bonding, general
 - CONTROL_OFF (TCL) E2.19 - 20, I3.11 - 12
 - CONTROL_ON (TCL) E2.19 - 20, I3.12 - 13
 - Conversions
 - See R/BASIC, formatting
 - See R/BASIC, strings
 - CONVERT (R/BASIC) N4.42 - 43
 - Coordinated locks
 - See File and record locking
 - See SQL, transaction processing
 - Copying text D4.6
 - COS (R/BASIC) N4.43 - 44
 - Cosine (R/BASIC) N4.43 - 44
 - COUNT (R/BASIC) N4.44 - 45
 - COUNT (SQL) K8.13
 - COUNT (TCL) I3.13 - 15
 - COUNT(*) (SQL) K8.13
 - CREATE TABLE (SQL) K8.34 - 36
 - CREATE VIEW (SQL) K8.36 - 37
 - CREATE-ACCOUNT (TCL)
 - See MAKEACCOUNT (TCL)
 - Cross Reference indexes F7.4 - 7
 - CRT
 - See Screen
 - CURRENT (SQL) K8.13
 - CURRENT OF (SQL) K8.14
 - Cursor (screen)
 - See also Data entry and editing
 - See also R/BASIC, data entry and output
 - See also R/BASIC, environment moving D3.1 - 5
 - CURSOR FOR (SQL) K8.14
 - Cursors (select lists)
 - See R/BASIC, cursors
 - See R/BASIC, files
 - See R/BASIC, system subroutines
 - Cursors (SQL)
 - accessing K7.8
 - activating K8.46
 - closing K7.8, 8.33
 - creating K7.8
 - defining K8.38
 - definition K7.6
 - positioning K7.6, 8.42 - 44
 - in R/BASIC subroutines K7.9
 - referencing queries K8.14
 - referencing variables assigned to K8.14
 - updating tables K7.8
 - Cutting and pasting text
 - See Editor
 - CW (TCL) I3.15
- ## D
- DATA (R/BASIC) N4.46 - 47
 - Data conversions F5.21, 5.27 - 32
 - See also R/BASIC, formatting
 - Data dictionaries
 - See Dictionaries
 - Data entry and editing
 - See also Chapters D2 and D3
 - See also Editor
 - See also R/BASIC, data entry and output
 - See also R/BASIC, system subroutines
 - accessing Query G4.1
 - browse list of records changed G1.5
 - keystrokes D1.5
 - preserving fields F5.37 - 38

- Data entry windows
 - See Paint
 - See Windows
- Data formatting F5.27 - 32
 - See also R/BASIC, formatting
 - See also R/BASIC, numbers
 - See also R/BASIC, reports
 - See also R/BASIC, strings
 - See also R/BASIC, system subroutines
- Data types E1.9 - 10
 - See also Environmental bonding, ASCII bond
 - See also Environmental bonding, dBASE III bond
 - See also Environmental bonding, general
- Data validation F5.19 - 27
 - See also R/BASIC, files
- Database servers L1.3
- Date (TCL) I3.87 - 88
- Date (SQL) K8.13
- DATE() (R/BASIC) N4.47 - 48
- dBASE III
 - See Environmental bonding, dBASE III bond
- DBL-SPC (R/LIST) J2.4
- Deadlocks K6.3 - 4, L3.10 - 12
- DEBUG (R/BASIC) N4.48
- Debugger
 - See R/BASIC, compiler
- Decision logic
 - See R/BASIC, branching and flow
- DECLARE (R/BASIC) N4.49
- DECLARE (SQL) K7.8, 8.38
 - See also Cursors (SQL)
- Declaring external functions (R/BASIC) N4.49
- DEFAULT (SQL) K8.14
- Default drive (R/BASIC) N4.57
- Defaults, system I3.72
- DEFINEFILE (TCL) I3.15 - 17
- DELETE (TCL)
 - See DELETEREC (TCL)
- DELETE (R/BASIC) N4.50 - 51
- DELETE (SQL) K7.8, 8.39 - 41
- DELETE() (R/BASIC) N4.51 - 53
- DELETE-ACCOUNT (TCL)
 - See DELETEACCOUNT (TCL)
- DELETEACCOUNT (TCL) I3.17 - 18
- DELETEFILE (TCL) I3.18 - 19
- DELETEREC (TCL) I3.19 - 20
- DELETEUSER (TCL) I3.20 - 21
- Delimiters N3.16 - 17
 - See also R/BASIC, dynamic arrays
 - See also R/BASIC, strings
- DELLIST (TCL) I3.21
- DET-SUPP (R/LIST) J2.4
- DETACH (TCL) I3.22
- DICT (TCL) I3.23
- Dictionaries
 - See also Chapter E3
 - See also Fields
 - See also Files
 - See also Files, bonded
 - See also Formulas
 - See also Paint, prompts
 - See also R/BASIC, files characteristics and format (list) E3.8 - 13
 - clearing (TCL) I3.7
 - defining and creating E2.2 - 3
 - definition E3.1
 - editing E3.3
 - editing (TCL) I3.23, 3.25
 - listing E3.7 - 8
 - listing (TCL) I3.38 - 39
 - names E3.2
- Dictionary files
 - See Dictionaries
- Dictionary formulas
 - See Formulas
- Dictionary types
 - See Fields
- DIMENSION (R/BASIC) N4.53 - 54
- Dimensioned arrays
 - See R/BASIC, matrices
- DIR() (R/BASIC) N4.54 - 56
- Directories
 - See R/BASIC, environment
 - See R/BASIC, operating system
 - See Volumes
- DIRLIST() (R/BASIC) N4.56 - 57
- Disk drive (R/BASIC) N4.57
- DISTINCT (SQL) K8.14
- Domain-level validation and conversion N3.42

Domain controls E2.19

DOS

See Operating system

See R/BASIC, operating system

Dragging prompts F4.19 - 20

DRIVE() (R/BASIC) N4.57

DROP (SQL) K8.15

DROP TABLE (SQL) K8.41

DROP VIEW (SQL) K8.41 - 42

DUMP (TCL) I3.23 - 24

Dynamic arrays

See R/BASIC, dynamic arrays

E

EasyWriter

See Reports, EasyWriter

ECHO (R/BASIC) N4.58

EDIT (TCL) I3.25

Edit masks F5.7

Editor

See also Chapter D4

See also Data entry and editing

See also R/BASIC, data entry and output

See also R/BASIC, system subroutines

accessing N1.2

accessing (R/BASIC) N5.32 - 41

accessing (TCL) I3.25

blocking text D4.5 - 8

buffers D4.5 - 8

checking record size D4.12

converting text to upper or lower case D4.12

copying (cutting and pasting) blocks D4.6

definition D3.1

deleting blocks D4.7

deleting records D3.9

displaying ASCII character codes D4.13

editing \$INSERT records D4.11

editing additional records D4.11

editing multivalues D4.10

editing subvalues D4.10

editing text lines D4.10

(*cont.*)

Editor (*cont.*)

environment D4.2 - 4

keystroke translator D4.13

markers used to separate field values D4.9

modes D3.8

moving blocks D4.7

operating system files D4.4, 4.12 - 13

saving and compiling programs D4.13

searching and replacing D4.8 - 9

shifting block indentation D4.12

softkeys D4.10 - 11

status D3.8

tab stops D4.2 - 3

zooming D3.10

Elements

See R/BASIC, dynamic arrays

See R/BASIC, matrices

Embedded SQL

See SQL, programming (embedded SQL)

ENCRYPT (TCL) I3.25 - 26

END (R/BASIC) N4.59

END CASE (R/BASIC) N4.27 - 28

Ending Advanced Revelation D1.2

Entering Advanced Revelation

See Logon

Entry

See Data entry and editing

Entry lines

See Paint, prompts

Environment

See also R/BASIC, operating system

changing characteristics F3.14

characteristics assigned at logon F3.17

consistency levels (SQL) K6.5, 8.12

creating F3.15

default data volume E4.5 - 6

definition F3.13

displaying current status (TCL) I3.89 - 90

Editor D4.2

global F6.1 - 2

indexing F7.7, 7.14

keystroke capture F11.4

(*cont.*)

Environment (cont.)

- locking and transaction
 - processing K6.1 - 6
- macros F11.1
- menu displayed at logon F6.3
- Paint F4.11, 5.53
- Query G4.11
- SQL K6.1 - 2
- SYSTEM file E2.29 - 30
- terminal emulation
 - settings (TCL) I3.84 - 85
- View window H1.6 - 7

Environmental bonding, general

- accessing files M1.7
- accounts M1.9
- bonded files vs. Advanced Revelation
 - files M1.8 - 9
- configuring bonds M1.5
- control files M1.10 - 12
- data loss M1.21 - 23
- data types F4.16, M1.17 - 20
- defining new files M1.6 - 7
- definition M1.1 - 4
- deleting files M1.16 - 17
- dictionaries M1.14 - 15
- displaying installed bonds M1.5
- field attributes M1.20 - 21
- file and record structure M1.12 - 13
- indexes M1.25 - 26
- locking files and records L3.13
- installing bonds M1.5
- naming fields and files M1.10
- networks and locking M1.26
- null handling M1.23 - 24
- options M1.8
- record keys M1.13 - 14
- restructuring files M1.16
- volumes M1.5 - 6

Environmental bonding, ASCII bond

- accessing files M2.3
- adding records M2.12
- characteristics M2.2 - 3
- copying files M2.11
- copying records M2.11
- defining dictionaries for existing
 - files M2.3 - 8

(cont.)

Environmental bonding, ASCII bond (cont.)

- defining new files M2.8 - 10
- deleting records M2.12
- indexing M2.13
- naming files M2.8 - 9
- networks and locking M2.14
- null handling M2.13
- variable vs. fixed-length files M2.1
- writing invalid data to fields M2.12

Environmental bonding, dBASE III bond

- accessing files M3.3 - 4
- adding records M3.7
- characteristics M3.2 - 3
- copying files M3.8
- copying records M3.7 - 8
- data types M3.5 - 6
- data types (table) M3.13 - 15
- defining new files M3.4 - 6
- deleting records M3.7
- dictionaries for existing files M3.4
- field attributes M3.6
- field positions M3.6
- increasing available file
 - handles M3.1 - 2
- indexing M3.9 - 12
- logon options M3.1
- naming fields M3.13
- naming files M3.12 - 13
- networks and locking M3.12
- rebuilding files M3.8 - 9
- writing invalid data to fields M3.6 - 7

EQU(ATE)(R/BASIC) N4.60 - 61

Equating symbols with variables,
 constants, or functions
 (R/BASIC) N4.60 - 61
 See also R/BASIC, constants
 See also R/BASIC, variables

Errors

See also R/BASIC, errors
 See also R/BASIC, system subroutines
 branching on (SQL) K8.23
 continuing execution after
 (SQL) K8.13
 testing for (SQL) K7.9 - 10
 testing for errors within modules
 (SQL) K8.26

EVAL (TCL) I3.27
 EW (TCL) I3.27
 Exclusive locks F5.34, L3.3
 EXECUTE (R/BASIC) N4.61 - 62
 Executing programs
 See R/BASIC, subroutines and functions
 EXISTS (SQL) K8.15
 EXIT
 See Operating system
 Exiting Advanced Revelation D1.2
 EXP (R/BASIC) N4.62 - 63
 Expanded memory
 See Chapter B5
 EXPENDABLE (R/BASIC) N4.63 - 64
 Explicit locking L3.6
 Exploding sorts J2.9 - 10
 Exponentiation N4.167 - 168
 Exponents (base e) N4.62 - 63
 EXPORT (TCL) I3.28
 External format F5.27 - 32
 External functions (R/BASIC) N3.28 - 29
 See also R/BASIC, subroutines and functions
 External subroutines (R/BASIC) N3.24
 See also R/BASIC, subroutines and functions
 EXTRACT (R/BASIC) N4.64 - 65

F

FETCH (SQL) K7.8, 8.42 - 44
 See also Cursors (SQL)
 FIELD (R/BASIC) N4.65 - 66
 Field marks (R/BASIC) N3.16 - 17
 Fields
 See also Chapter F8
 See also Columns (SQL)
 See also Columns, multivalued (SQL)
 See also Dictionaries
 See also Environmental bonding
 See also Indexes
 See also Paint, prompts
 See also R/BASIC, fields
 @CRT E3.6
 @LPTR E3.6
 (cont.)

Fields (cont.)

 associated multivalues F5.2
 attributes E1.8 - 9, E2.12
 copying in windows D3.9
 data types E1.9 - 10
 defining multipart keys E2.9 - 10
 defining synonyms E2.10 - 11
 defining type G (Group) E2.11 - 12
 defining type S (Symbolic) E2.8 - 10, N3.51 - 52
 defining E2.4 - 6
 definition E1.2
 dictionary characteristics and format (list) E3.8 - 12
 editing E3.3
 editing attributes (TCL) I3.23
 markers used to separate field values D4.9
 maximum number and size E1.6
 multivalued (definition) E1.7
 naming conventions E1.5
 naming in bonded files F4.9
 positions E2.4
 positions in bonded files F5.17
 summing data (TCL) I3.80 - 82
 synonyms E1.8, 3.4 - 5
 temporary E3.6
 type F (Field) E3.3 - 4
 type G (Group) E1.7 - 8, 3.6
 type S (Symbolic) E1.8, 3.4 - 5
 variable-length E1.6
 zooming D3.10
 FIELDSTORE (R/BASIC) N4.67 - 68
 File and record locking
 See also Chapter L3
 See also R/BASIC, files
 ASCII environmental bond M2.3
 automatic locking L3.5 - 6
 bonded environments L3.13, M1.26
 coordinated locks F5.35 - 36, L3.4
 displaying locking information (TCL) I3.43 - 44
 exclusive locks F5.34, L3.3
 explicit locking L3.6
 file locking L3.3
 file types L2.2
 (cont.)

File and record locking (*cont.*)

- global coordinated locking L3.8
- implicit locking in SQL K6.5 - 6, 8.12, 8.55 - 56, L3.8 - 13
- limiting available locks L3.7
- lock table L3.5
- multiple file servers L3.13
- networks F5.34 - 36, L1.5
- record locking L3.3
- record locking in windows F5.34 - 36
- shared locks F5.35, L3.3 - 4
- transaction processing E2.26 - 27, L3.7
- user-optional locks L3.6

File keys

- See Record keys

File locking

- See File and record locking

File servers L1.3, L3.13**FILECOPY (TCL) I3.29 - 30****Files**

- See also Batch updates
 - See also Chapters E1 and E2
 - See also Dictionaries
 - See also Environmental bonding
 - See also Fields
 - See also File and record locking
 - See also Indexes
 - See also Paint, general
 - See also R/BASIC, cursors
 - See also R/BASIC, files
 - See also R/BASIC, operating system
 - See also System files
 - See also Tables (SQL)
 - See also Transaction processing
 - See also Volumes
 - accessing (TCL) I3.1, 3.22
 - accessing with LOGTO (TCL) I3.43 - 44
 - accessing with SQL K2.9
 - attaching E4.2, I3.1
 - attributes E2.13 - 14
 - clearing (TCL) I3.7
 - copying (TCL) I3.29 - 30
 - creating (TCL) I3.15 - 17
 - creation sequence E3.2
- (*cont.*)

Files (*cont.*)

- defining and creating
 - dictionaries E2.2 - 3
 - defining multipart keys E2.9 - 10
 - defining E2.1 - 2
 - definition E1.1
 - deleting (TCL) I3.18 - 19
 - detaching E4.3, I3.22
 - displaying locking information (TCL) I3.42 - 43
 - examining Linear Hash (TCL) I3.23 - 24
 - exporting (TCL) I3.28
 - file types on networks L2.2
 - flushing buffers (TCL) I3.31
 - global (list) E2.29
 - importing (TCL) I3.33
 - Linear Hash E1.2 - 3, 2.13 - 14
 - listing attached (TCL) I3.39 - 40
 - listing files in a volume (TCL) I3.41 - 42
 - naming conventions E1.5
 - redefining (TCL) I3.59 - 60
 - renaming (TCL) I3.61 - 62
 - resizing (TCL) I3.59 - 60
 - ROS E1.3, E2.14
 - routing printer output to (TCL) I3.54 - 55
 - size E1.5
 - synonyms E2.15 - 16, I3.72 - 74
 - types E1.2 - 3
- Files, bonded**
- See also Environmental bonding, ASCII bond
 - See also Environmental bonding, dBASE III bond
 - accessing with SQL K2.9
 - data types E2.7 - 8, F4.11, 4.16
 - defining E2.6 - 7
 - editing specifications F4.10, 4.16
 - editing E2.8
 - field attributes E2.12 - 13
 - field positions F5.17
 - naming fields F4.9
 - naming E2.6
 - on networks L2.2
 - Painting F4.10
 - volumes E4.4

Filing system
 See Files
 Filters J1.6 - 14
 See also R/BASIC, cursors
 See also Selected lists
 FILTER (TCL) I3.30
 Filter key
 See Chapter G2
 Flow
 See R/BASIC, branching and flow
 FLUSH (R/BASIC) N4.68 - 69
 FLUSH (TCL) I3.31
 FMT (R/BASIC) N4.69 - 70
 FOOTING (R/BASIC) N4.71 - 72
 FOOTING (R/LIST) J2.11 - 12
 Footings
 See R/BASIC, reports
 See Reports, EasyWriter
 See Reports, R/LIST
 FOR ... NEXT (R/BASIC) N4.72 - 74
 Forcing page eject
 See Page eject
 Foreign environments
 See Environmental bonding and
 specific topics
 FORM (TCL) I3.31 - 32
 Form feed
 See Page eject
 Form parameters H2.7 - 9
 FORM-LIST (TCL)
 See MAKELIST (TCL)
 Formatting data F5.27 - 32
 See also R/BASIC, formatting
 See also R/BASIC, numbers
 See also R/BASIC, strings
 See also R/BASIC, system subroutines
 Formatting reports
 See R/BASIC, reports
 See Reports, EasyWriter
 See Reports, forms
 See Reports, Merge
 See Reports, R/LIST
 See SQL, queries
 Forms
 See Reports, forms

Formulas (R/BASIC)
 See also Chapter F8
 See also R/BASIC topics
 See also Reports, R/LIST
 creating F8.1 - 3, N3.51 - 52
 glossary of trigger words F8.4 - 8
 types F8.3 - 4
 FROM (SQL) K8.15
 FMSG (R/BASIC) N5.13 - 14
 Full-screen Editor
 See Editor
 FUNCTION (R/BASIC) N4.74 - 76
 Functions
 See R/BASIC, branching and flow
 See R/BASIC, subroutines and
 functions
 See R/BASIC, system subroutines

G

GARBAGECOLLECT (R/BASIC) N4.76
 GET-LIST (TCL)
 See GETLIST (TCL)
 GET.CURSOR (R/BASIC) N4.78
 GETBREAK (R/BASIC) N4.77
 GETECHO (R/BASIC) N4.77
 GETLIST (TCL) I3.32
 GETPRINTER (R/BASIC) N4.77
 GETPROMPT (R/BASIC) N4.77
 Global files (list) E2.29
 GOSUB (R/BASIC) N4.79 - 80
 GOSUB (SQL) K8.15
 GOTO (R/BASIC) N4.80 - 81
 GOTO (SQL) K8.16
 GRAND-TOTAL (R/LIST) J2.12 - 13
 Group (type G) fields
 See Fields
 GROUP BY (SQL) K8.16
 Grouping (associating) multivalued
 fields F5.2
 Grouping records
 See Reports, R/LIST
 See SQL, queries

H

HASH (R/BASIC) N4.81 - 82
 Hash algorithm (ROS files) N4.81 - 82
 HAVING (SQL) K8.16 - 17
 HDR-SUPP (R/LIST) J2.4
 HEADING (R/BASIC) N4.82 - 83
 HEADING (R/LIST) J2.13 - 14
 Headings
 See R/BASIC, reports
 See Reports, EasyWriter
 See Reports, R/LIST
 Help
 adding in Paint F5.38
 definition D2.2 - 3
 for popups F6.12
 level set at logon B4.2
 protecting F5.37
 status line D2.3 - 4
 types available (table) D1.4
 Hexadecimal values (R/BASIC)
 constants N3.4
 conversions N4.90 - 91, 4.140 - 141
 Horizontal multivalues F4.12

I**I/O**

See R/BASIC, files

ICONV (R/BASIC)

See also R/BASIC, system subroutines

B (Boolean) conversions N4.85 - 86
 converting data to internal
 format N4.83 - 85
 D (date) conversions N4.86 - 88
 DT (datetime) conversions N4.88 - 90
 MD (masked decimal)
 conversions N4.92 - 93
 MS (scientific notation)
 conversions N4.93 - 94
 MT (time) conversions N4.94 - 96
 MX, HEX, MO, and MB (binary,
 hex, octal) conversions N4.90 - 91

ID-SUPP (R/LIST) J2.4

IF (R/BASIC) N4.96 - 99

Implicit locking

See SQL, general

See SQL, transaction processing

IMPORT (TCL) I3.33

IN (SQL) K8.17

INDEX (R/BASIC) N4.99 - 100

INDEX (TCL) I3.33 - 34

INDEX.FLUSH (R/BASIC) N5.15 - 16

Indexes

See also Chapters F7 and G3

See also Environmental bonding

See also R/BASIC, system subroutines

background processing F7.13 - 14

Btree F7.3 - 4

creating browse lists G3.2 - 5

Cross Reference F7.4 - 7

definition E1.6

deleting F7.21 - 22

deleting indexed files F7.14

disk storage space requirements F7.2

global files F7.9

index transaction code F7.22 - 24

listing F7.16 - 17

listing (TCL) I3.40

maintenance F7.13 - 15

moving indexed files F7.15

networks and F7.2

Quickdex/Rightdex F7.10 - 12

rebuilding F7.19

Relational F7.7 - 9

reserved characters G3.5

searching in application

 windows G3.2 - 5

stop lists F7.4 - 5

updating F7.18 - 21

updating (TCL) I3.33 - 34

INIT.VIEW (R/BASIC) N5.16 - 18

INITDIR (R/BASIC) N4.100

Initializing Advanced Revelation I3.62

INITRND (R/BASIC) N4.101 - 102

INMAT() (R/BASIC) N4.102 - 103

INP() (R/BASIC) N4.103

Input

See R/BASIC, data entry and output

INPUT() (R/BASIC) N4.104 - 105

INPUT.CHAR (R/BASIC) N5.18 - 19

INSERT (R/BASIC) N4.105 - 107

INSERT (SQL) K8.44 - 45

Installation

See also Chapters B1, B2, and B3
 basic system B1.2 - 3
 files affected B2.9 - 10
 on networks B2.2, 2.4
 optional modules, existing
 systems B2.1 - 3
 optional modules, new
 systems B1.3 - 5
 removing optional modules B3.1
 requirements B1.1
 security restrictions B2.9, 2.11
 updates and upgrades B2.3 - 6
 upgrading user accounts B2.6 - 10

INT (R/BASIC) N4.107 - 108

Integers

See R/BASIC, numbers

Internal functions

See R/BASIC, subroutines and
 functions

Internal subroutines

See R/BASIC, subroutines and
 functions

INTO (SQL) K8.17

Inverse (negative) (R/BASIC) N4.129

J**Joining**

fields with other files F5.9 - 10
 tables (SQL) K4.11 - 12
 windows (relating) F5.39 - 40

Justification F4.17

JUSTLEN (R/LIST) J2.15

K

KEY (SQL) K8.22

Key columns (SQL)

See also Record keys
 defining K3.4 - 5, 8.22
 documenting foreign keys K8.22
 multivalued K8.17 - 18
 multivalued columns (repeating
 rows) K2.10

Key fields

See Record keys

KEYQUAL (SQL) K8.17 - 18

Keystroke capture

See Chapter F11

Keystroke translator D4.13

Keystrokes

See also R/BASIC, system subroutines
 disabling F5.43
 table D1.5

L

LABEL (TCL) I3.34 - 35

Labeled COMMON

(R/BASIC) N4.41 - 42

Labels

See Paint, labels

See Reports, labels

Labels, smart F5.11

LANs

See Networks

LATENT (R/LIST) J1.13

Latent select lists

See R/BASIC, cursors

See Select lists

LEN (R/BASIC) N4.109 - 110

Length of strings

(R/BASIC) N4.109 - 110

LIKE (SQL) K8.18

LIMIT (R/LIST) J2.16 - 17

Linear Hash files E1.2 - 3, 2.13 - 14

LINEMARK (R/BASIC) N4.110

Linking subroutines and functions

See R/BASIC, branching and flow

See R/BASIC, compiler

See R/BASIC, subroutines and
 functions

LIST (TCL) I3.35 - 37

See also SORT (TCL)

LIST ONLY (R/LIST)

See ONLY (R/LIST)

LISTDICT (TCL) I3.38 - 39

LISTFILES (TCL) I3.39 - 40

LISTINDEX (TCL) I3.40

LISTMEDIA (TCL)

See LISTVOL (TCL)

LISTVOL (TCL) I3.41 - 42

LISTVOLUMES (TCL) I3.42

LN (R/BASIC) N4.110 - 111

- LO (R/LIST)
 - See ONLY (R/LIST)
- Local Area Networks
 - See Networks
- LOCATE (R/BASIC) N4.111 - 113
- LOCATE BY (R/BASIC) N4.113 - 114
- LOCK (R/BASIC) N4.114 - 117
- Locking
 - See File and record locking
 - See R/BASIC, files
 - See SQL, general
- LOCKS (TCL) I3.42 - 43
- Log to
 - See Security
- Logarithm (R/BASIC) N4.110 - 111
- Logoff D1.2
- LOGOFF (TCL) I3.43
- Logon
 - See also Chapter B4
 - See also Chapter D1
 - expanded memory logon
 - options B5.2 - 4
 - logon scripts on networks L2.5
 - logon verb F3.10
 - networks L2.2 - 3
 - options F3.1
 - options with dBASE III bond M3.1
- LOGTO (TCL) I3.43 - 44
- LOOP (R/BASIC) N4.117 - 119
- Looping
 - See R/BASIC, branching and flow
- Lower/upper case conversions
 - Editor D4.12
 - R/BASIC N4.43
- LPTR (R/LIST) J2.4

- M**
- MACRO (TCL) I3.44 - 45
- Macro keys
 - See also Chapter F11
 - See also Softkeys
 - activating (TCL) I3.44 - 45
- Maintenance updates
 - See Installation
- MAKEACCOUNT (TCL) I3.45 - 46
- MAKEFILE (TCL)
 - See DEFINEFILE (TCL)
- MAKELIST (TCL) I3.46 - 47
- MAKEUSER (TCL) I3.47
- MAKEVOC (TCL) I3.48 - 49
- Masking (R/BASIC) N4.69 - 70
- MAT (R/BASIC) N4.119 - 121
- MATCHES (R/BASIC) N4.121 - 122
- Matching patterns
 - (R/BASIC) N4.121 - 122
- Math
 - See R/BASIC, numbers
- MATPARSE (R/BASIC) N4.122 - 123
- MATREAD (R/BASIC) N4.123 - 125
- Matrices
 - See R/BASIC, matrices
 - See R/BASIC, variables
- MATUNPARSE
 - (R/BASIC) N4.125 - 126
- MATWRITE (R/BASIC) N4.126 - 128
- MAX (SQL) K8.18 - 19
- Memory management **MEMSPACE**
 - See Chapter B5
 - See R/BASIC, memory management
- Menus
 - See also Chapter F6
 - See also R/BASIC, system subroutines
 - accessing Main Menu from
 - windows D2.10
 - creating F6.2
 - displayed at logon F6.3
 - displaying (TCL) I3.66 - 67
 - keystrokes D2.6 - 7
 - moving on the screen D2.13
- Merge
 - See Reports, Merge
- MERGE (TCL) I3.49 - 50
- Merging programs
 - See R/BASIC, compiler
- Messages (R/BASIC) N5.19 - 22
- MIN (SQL) K8.19
- MOD (R/BASIC) N4.128 - 129
- MODIFY (SQL) K8.19
- Modules
 - See R/BASIC, branching and flow
 - See R/BASIC, subroutines and
 - functions
- Modulo (remainder) (R/BASIC) N4.128
- Moving text
 - See Editor

MS-DOS

See Operating system

See R/BASIC, operating system

MSG (R/BASIC) N5.19 - 22

Multivalued arithmetic

(R/BASIC) N3.8, 4.9 - 10

Multivalued fields E1.7

See also Paint, prompts

See also R/BASIC, dynamic arrays

See also R/BASIC, numbers

See also Columns, multivalued (SQL)

N

NAMEMEDIA (TCL)

See NAMEVOLUME (TCL)

NAMEVOLUME (TCL) I3.50 - 51

Naming conventions

ASCII files M2.8 - 9

bonded fields and files M1.10

data entry windows F4.3

database terminology (table) E1.2

dBASE III fields M3.13

dBASE III files M3.12 - 13

dictionaries E3.2

fields and files E1.5

fields in bonded files F4.9

indexing files F7.10

Merge templates H6.3

popups F6.4 - 5

SQL columns, tables, and

views K2.5 - 6, 2.8 - 9

SQL tables, temporary K4.12

volumes E4.4

Natural logarithm

(R/BASIC) N4.110 - 111

NEG (R/BASIC) N4.129 - 130

Negative (inverse) (R/BASIC) N4.129

Nesting queries (SQL) K4.13 - 14

Networks

See also Chapters L1 and L2

See also File and record locking

See also SQL, transaction processing

accessing files L2.1

(*cont.*)

Networks (*cont.*)

accessing network command

processor L2.5 - 6

bonded files L2.2

bumping Advanced Revelation I3.7,

L1.4

compatibility of Advanced

Revelation with L1.3 - 4

creating and deleting files L2.1 - 2

database servers L1.3

definition L1.1

displaying current status

(TCL) I3.89 - 90

displaying locking information

(TCL) I3.42 - 43

drivers L1.4

environment settings L2.4 - 5

file servers L1.3

file types L2.2

hardware L1.2

increasing number of users

(TCL) I3.7, L1.4

indexing files F7.2, L2.6 - 8

installing Advanced Revelation B2.11

locking files and records L1.5

logging on L2.2 - 3

logon scripts L2.5

network information report L1.6

printing L2.6

R/BASIC programming L2.9 - 10

requirements L2.2 - 3

software L1.2

station number L1.6, 2.10

transaction processing L2.8 - 9

users vs. workstations L1.6

users L2.4 - 5

video settings L2.5

writing applications for L1.5

Non-terminating cursors

See R/BASIC, cursors

NOPAGE (R/LIST) J2.4

NORESOLVE (R/LIST) J1.13

Normalizing repeating data (SQL)

See Columns, multivalued (SQL)

NOT (R/BASIC) N4.130 - 131

NOT NULL (SQL) K8.20

NOTFOUND (SQL) K8.19

Null/null handling

See also R/BASIC, compiler

See also R/BASIC, numbers

See also R/BASIC, strings

ASCII environmental bond M2.13

dynamic arrays (R/BASIC) N3.5 - 6

environmental bonding M2.13

null statements (R/BASIC) N4.132

SQL K2.10 - 11

NULL (R/BASIC) N4.132

NULL (SQL) K8.20

NUM (R/BASIC) N4.133 - 134

Numeric precision

See R/BASIC, numbers

O

Object code (R/BASIC) N3.1

OCONV (R/BASIC)

See also R/BASIC, system subroutines

B (Boolean) conversions N4.135 - 136

converting data to output

format N4.134 - 135

D (date) conversions N4.136 - 138

DT (datetime)

conversions N4.138 - 140

MD (masked decimal)

conversions N4.142 - 144

MS (scientific notation)

conversions N4.144 - 145

MT (time) conversions N4.145 - 146

MX, HEX, MO, and MB (binary, hex,

octal) conversions N4.140 - 141

OCONV (R/LIST) J2.18

Octal conversions (R/BASIC) N4.90 - 91,
4.140 - 141**OFF**

See LOGOFF (TCL)

OFF (TCL) D1.2

ON (SQL) K8.20

ON GOSUB (R/BASIC) N4.146 - 148

ON GOTO (R/BASIC) N4.148 - 149

Online help

See Help

Online tutorial

See Tutorial

ONLY (R/LIST) J2.18 - 19

OPEN (R/BASIC) N4.149 - 150

OPEN (SQL) K7.8, 8.46

See also Cursors (SQL)

Operating system

See also R/BASIC, environment

See also R/BASIC, files

See also R/BASIC, operating system

accessing D1.3

accessing (TCL) I3.53 - 54, 3.82 - 83

editing files D4.4, 4.12 - 13

editing files (TCL) I3.25

naming rollout file (TCL) I3.65

returning to Advanced

Revelation D1.3

Operators

See R/BASIC, dynamic arrays

See R/BASIC, logical expressions

See R/BASIC, numbers

See R/BASIC, strings

OR (R/BASIC) N4.20 - 21

Orphans (Reports, Merge) H6.5

ORDER BY (SQL) K8.21

OS/2

See Operating system

See R/BASIC, operating system

OSBREAD (R/BASIC) N4.150 - 152

OSBWRITE (R/BASIC) N4.152 - 153

OSCLOSE (R/BASIC) N4.153 - 154

OSDELETE (R/BASIC) N4.154 - 155

OSOPEN (R/BASIC) N4.155 - 156

OSREAD (R/BASIC) N4.156 - 157

OSWRITE (R/BASIC) N4.158

OUT (R/BASIC) N4.159

Output

See R/BASIC, data entry and output

See Reports

Output conversions F5.28 - 30

See also OCONV (R/BASIC)

Output format F5.27 - 32

P

PAGE (R/BASIC) N4.160

Page eject

forcing, Merge reports H6.10

forcing, R/BASIC reports N4.160

forcing, R/LIST reports J2.8

suppressing, R/BASIC program

listing (TCL) I3.5

suppressing, R/LIST reports J2.4

Page headings

See R/BASIC, reports

See Reports, EasyWriter

See Reports, R/LIST

PAINT (TCL) I3.51 - 52**Paint, Forms**

See Reports, Forms

Paint, general

See also Chapters F4, F5, and H2

accessing F4.2

binding methods F4.8 - 9, 5.14 - 15

bonded files F4.10

creating data files F4.6

deleting dictionary fields F5.16

deleting windows F4.5 - 6

displaying a list of active keys F4.25

displaying the Paint Main

Menu F4.25

editing dictionary fields F5.16

editing modes F4.24 - 25

Editor F4.23

field positions in bonded files F5.17

listing dictionary fields F5.16

moving the cursor F4.13

options F4.23

panning through virtual space F4.28

Quick Paint F5.32 - 33

Roam mode F4.23

softkeys F4.23 - 24

status line F4.7 - 8

Paint, labels

definition F4.21 - 22

smart labels F5.11

Paint, prompts

See also Fields

associating multivalues

(grouping) F5.2

characteristics (list) F5.5 - 7

color and video display F4.19

converting to labels F4.20

copying characteristics F5.3 - 4

creating F4.8 - 12, 5.32 - 33

creating formulas F5.11 - 12

data types F4.11 - 12, 4.15 - 16

data validation F5.19 - 27

default characteristics F5.3

(*cont.*)

Paint, prompts (*cont.*)

default values F5.6 - 7

definition F4.1

dictionary fields and F4.15

dragging F4.19 - 20

edit masks F5.7

editing F4.14, 5.10

entry lines F4.1, 4.12

entry type options (table) F4.17 - 18

field vs. prompt characteristics F4.8,

4.14, 5.13, 5.18 - 19

formatting data F5.27 - 32

help F5.38

horizontal multivalues F4.12

joining fields with other files F5.9 - 10

justification F4.17

labels F4.2

length F4.16

limiting entry length F5.7

limiting rows in multivalues F5.7

multivalued F4.12 - 13, 5.6

naming fields in bonded files F4.9

options F4.18

processing F5.8 - 9

Prompt Table F5.10

Prompt Window F5.5

record keys F4.10, 5.6

registers F5.51 - 52

removing F4.20

reordering F5.2 - 3

tab stops F5.6

Table mode F5.7 - 8

Paint, reports

See Reports, forms

Paint, windows

See also Windows

See also R/BASIC, system

subroutines

border types F5.34

collector windows F4.4, 5.43 - 47

creating softkeys F5.39

data entry windows F4.3 - 4

disabling keys F5.43

editing existing F4.3

exploding F5.36

help F5.38

(*cont.*)

Paint, windows (cont.)

- joining fields with other files F5.9 - 10
- menu displayed at [F10] F5.36
- moving F4.27
- naming conventions F4.3
- page tabs F4.27
- panning through virtual space F4.28
- preserving fields F5.37 - 38
- processing F5.47 - 52
- protecting help F5.37
- Query F5.55 - 57
- R/LIST F5.52 - 54
- recalculating symbolic fields F5.12 - 13
- recalculating symbolic fields based on dependency F5.54 - 55
- record locking F5.34 - 36
- registers F5.51 - 52
- relating F5.39 - 41
- reordering prompts F5.2 - 3
- resizing F4.26 - 27
- saving F4.5
- security F5.41 - 42
- start prompt F5.37
- tab stops F5.6
- Table mode F5.7 - 8
- testing F4.6, 4.29
- video attributes F5.41
- virtual space F4.5, 4.28

PAINT.FORM (TCL) I3.51 - 52**Panning D2.14 - 15, 2.17****Parameters**

- See R/BASIC, subroutines and functions

- See R/BASIC, variables

Parentheses (R/BASIC) N3.8, 3.27

- See also R/BASIC, subroutines and functions

PASSWORD (TCL) I3.52**Passwords**

- See Security

Pattern matches

- at data entry F5.19 - 20
- R/BASIC strings N4.121 - 122

PC

- See Operating system

PC (TCL) I3.53 - 54

- See also SUSPEND (TCL)

PCPERFORM (R/BASIC) N4.160 - 161**PDISK (TCL) I3.54 - 55****PERFORM (R/BASIC) N4.161 - 162****POP.UP (R/BASIC) N5.22 - 28****Popups**

- See also Chapter F6

- See also R/BASIC, system subroutines

- creating help F6.12

- data displayed F6.7 - 10

- data returned F6.12 - 13

- displaying with codes and commands F6.13 - 14

- EasyWriter H5.4 - 5

- for validation patterns F5.26 - 27

- format F6.10 - 12

- moving on the screen D2.13

- number and order of selections F6.5 - 7

- screen locations F6.5

- status line D2.18

- types H5.4 - 5

- using D2.18 - 20

Port

- See R/BASIC, operating system

PORT (TCL) I3.55**Preprocessor**

- See R/BASIC, compiler

(PRIMARY) KEY (SQL) K8.22**Primary keys**

- See Key columns (SQL)

- See Record keys

PRINT (R/BASIC) N4.162 - 164**PRINTER (R/BASIC) N4.164 - 165****TO PRINTER (SQL) K8.24****Printers**

- See also R/BASIC, data entry and output

- See also R/BASIC, environment

- displaying and editing current

- settings (TCL) I3.71

- Merge report specifications H6.6, 6.22 - 23

- printing forms H2.9 - 10

- printing labels H3.3 - 4

- printing Merge reports H6.19 - 21

- printing on networks L2.6

- (cont.)

Printers (cont.)

- printing R/BASIC programs I3.4 - 6
- printing R/LIST reports H4.3 - 4
- printing SQL tables K8.24
- routing output to files
(TCL) I3.54 - 55

Product upgrades

- See Installation

Programs

- See R/BASIC
- See SQL, programming (embedded SQL)

PROMPT (R/BASIC) N4.165 - 166

- Prompt character
(R/BASIC) N4.165 - 166

Prompts

- See Data entry and editing
- See Paint, prompts

PUT.CURSOR (R/BASIC) N4.166 - 167**PWR (R/BASIC) N4.167 - 168****Q****Qfiles**

- See Files, synonyms

Queries (SQL)

- See SQL, queries

Query (window searches)

- See also Chapter G4
- accessing D2.12, G4.1
- accessing (TCL) G4.9, I3.56
- accessing indexes G4.7 - 9
- border prompting G4.12
- creating G4.1 - 2
- customizing in Paint F5.55 - 57
- customizing selection editor G4.11 - 12
- displaying last G4.7
- editing SELECT commands G4.6
- effects on prompts G4.7
- environment G4.11 - 12
- executing G4.2
- masking characters G4.12
- operators G4.3
- saving as filters G4.6
- selection criteria G4.2 - 5
- sort criteria G4.5 - 6
- syntax G4.10 - 11

QUERY (TCL) I3.56**Quick Paint F5.32 - 33****Quickdex indexes F7.10 - 12****Quotation marks**

- (R/BASIC) N4.168 - 169

QUOTE (R/BASIC) N4.168 - 169**R****R/BASIC, general**

- accessing TCL commands I1.6
- adding programs to VOC file I3.48 - 49
- case sensitivity N3.1
- compiling and executing source code
(TCL) I3.27
- compiling source code (TCL) I3.9 - 11
- creating and executing
programs N1.2 - 3
- creating programs with Formula
window F8.1 - 3
- definition N1.1 - 2
- deleting programs in VOC
file I3.88 - 89
- executing compiled programs
(TCL) I3.65 - 66
- labeling statements N3.12 - 13
- listing programs N1.3
- listing programs (TCL) I3.4
- object code N3.1
- source code N3.1
- statements N3.1
- storing programs N3.1
- symbol table N3.1

R/BASIC, branching and flow

- See also R/BASIC, subroutines and functions

- aborting to TCL 4.17
- branching to internal
subroutines N4.79
- branching to statements N4.80 - 81
- canceling program execution N4.17,
4.59, 4.194 - 195
- codes and commands N5.7 - 10
- conditional branching N3.14 - 15,
4.27 - 28, 4.96 - 99, 4.146 - 149
- ending programs or blocks of
statements N4.59

(cont.)

R/BASIC, branching and flow (*cont.*)
 executing TCL commands N4.61 - 62
 external subroutines N4.26 - 27
 internal subroutines N4.80
 labeling statements N3.12 - 13
 looping N4.72 - 74, 4.117 - 119
 unconditional branching N4.29

R/BASIC, compiler
 canceling program execution N4.17,
 4.59, 4.194 - 195
 comments N4.1 - 2
 compiling code N1.2 - 3
 debugger N3.44 - 50, 4.48, 4.110
 ending programs or blocks of
 statements N4.59
 including external blocks of
 code N5.2 - 3
 inserting source code N4.7 - 8
 multiple statement operator N4.2
 null statements N4.132
 object code N1.2
 quotation marks N3.3 - 4
 records generated by N3.1
 source code N1.2
 statements N3.1
 substituting symbols for
 functions N4.60 - 61
 symbol table N3.1

R/BASIC, constants
 See also R/BASIC, numbers
 See also R/BASIC, strings
 See also R/BASIC, variables
 assigning N4.11 - 15
 character strings N3.4
 definition N3.3
 hex values N3.4
 length limit N3.4
 numeric N3.3 - 4
 quotation marks N3.3 - 4
 scientific notation N3.3
 substituting symbols for N4.60 - 61
 system N5.2 - 3

R/BASIC, cursors
 bi-directional N3.42 - 43
 clearing N4.34 - 35
 default cursor N3.41
 (*cont.*)

R/BASIC, cursors (*cont.*)
 definition N3.40 - 41
 file assignments N3.44
 initializing N5.28 - 32
 latent vs. resolved select
 lists N3.43 - 44
 reading next record N4.171 - 175
 select lists N3.31 - 32, 3.40
 selecting record keys N4.184, 5.28 - 32
 selecting record keys
 (sorted) N4.185 - 188
 sorted select lists N3.31
 terminating and
 non-terminating N3.43

R/BASIC, data entry and output
 character display on/off N4.58
 clearing the screen N4.16 - 17
 cursor position and shape N4.78
 cursor position on the
 screen N4.16 - 17
 cursor shape and location N4.166 - 167
 displaying screen messages N5.19 - 22
 Editor N5.32 - 41
 input from a hardware port N4.103
 output to a hardware port N4.159
 page eject N4.160
 printer on/off N4.164 - 165
 printing to printer or
 screen N4.162 - 164
 prompt character N4.77, 4.165 - 166
 prompting for data N4.104 - 105,
 5.18 - 22
 View window N5.16 - 18

R/BASIC, debugger
 See R/BASIC, compiler

R/BASIC, dynamic arrays
 See also R/BASIC, matrices
 See also R/BASIC, strings
 adding elements N3.18
 arithmetic operators N4.9 - 10
 assigning values to
 variables N3.17 - 18
 assigning matrix
 values to N4.125 - 126
 assigning values to
 matrices N4.122 - 123
 (*cont.*)

R/BASIC, dynamic arrays (cont.)

- concatenation N4.9 - 10
- definition N3.15 - 16
- deleting elements N3.19, 4.51 - 53
- delimiters N3.16 - 17, 3.20 - 21
- extracting fields, values, or subvalues N4.64 - 65
- extracting substrings using delimiters N4.178 - 179
- field, value, and subvalue positions N4.113 - 114
- inserting fields, values, or subvalues N4.105 - 107
- locating strings N3.18
- matrices vs. N3.21 - 22
- multiplying two arrays N3.20
- null handling N3.5 - 6
- records N3.16
- replacing data in elements N3.19 - 20
- replacing substrings N4.179 - 181
- sorting elements N3.20
- string extraction and assignment N4.10 - 11
- subscripts N3.17
- substring positions N4.111 - 113
- summing elements N4.198 - 199

R/BASIC, environment

- Advanced Revelation serial number N4.189 - 190
- break key on/off N4.24, 4.77
- character display on/off N4.58, 4.77
- creating and displaying popups N5.22 - 28
- cursor position and shape N4.78, 4.166 - 167
- debugger N3.44 - 50, 4.48, 4.110
- default drive N4.57
- displaying screen messages N5.19 - 22
- operating system files N4.54 - 57
- printer on/off N4.77, 4.164 - 165
- prompt character N4.77, 4.165 - 166
- saving, restoring, and clearing screen N5.47 - 49
- screen borders N5.5
- system date N4.47 - 48

(cont.)

R/BASIC, environment (cont.)

- system time N4.203 - 204
- system time and date N4.204 - 205
- writing to the status line N5.41 - 47

R/BASIC, errors

- file access N3.36
- file or record access N5.13 - 14
- operating system N3.38
- testing for N4.193 - 194
- types N3.35
- unassigned variables N3.2

R/BASIC, fields

See also Fields

See also R/BASIC, dynamic arrays

See also R/BASIC, strings

- accessing N3.50 - 51, 4.6 - 7, 4.24 - 25
- assigning matrix values to N4.126 - 128
- assigning values to matrices N4.123 - 125
- assigning values to variables N4.177 - 178
- creating dictionary formulas N3.51 - 52
- deleting from dynamic arrays N4.51 - 53
- dynamic arrays N3.16 - 22
- extracting fields from files N4.213 - 215
- extracting from dynamic arrays N4.64 - 65
- inserting into dynamic arrays N4.105 - 107
- positions in substrings N4.113 - 114

R/BASIC, files

See also Files

See also R/BASIC, cursors

See also R/BASIC, errors

See also R/BASIC, fields

See also R/BASIC, matrices

See also R/BASIC, operating system

See also R/BASIC, records

See also R/BASIC, system subroutines

- accessing N3.29 - 30

- accessing fields N4.6 - 7, 4.24 - 25

(cont.)

R/BASIC, files (*cont.*)

- Btree indexes N5.6 - 7, 5.10 - 11, 5.15 - 16
- clearing N4.33 - 34
- clearing cursors N4.34 - 35
- closing N3.30
- closing operating system
 - files N4.153 - 154
- control features N3.42
- deleting operating system
 - files N4.154 - 155
- domain-level validation and conversion N3.42
- errors N3.35 - 37
- hash algorithm (ROS files) N4.81 - 82
- locking N3.32 - 34, 4.114 - 117, 4.208 - 210
- opening N3.30, 4.149 - 150
- opening operating system
 - files N4.155 - 156
- reading from N4.169 - 171, 4.175 - 178, 4.213 - 215
- reading next record N4.171 - 175
- reading operating system
 - files N4.150 - 152, 4.156 - 157
- record keys N3.30 - 31
- translating N4.213 - 215
- updating N3.30
- writing fields to N4.212 - 213
- writing records to N4.210 - 212
- writing to operating system
 - files N4.152 - 153, 4.158

R/BASIC, formatting

- See also R/BASIC, reports
- See also R/BASIC, strings
- binary conversions N4.90 - 91, 4.140 - 141
- Boolean conversions N4.85 - 86, 4.135 - 136
- character conversions N4.90 - 91, 4.140 - 141
- converting data to internal format N4.83 - 85
- converting data to output format N4.134 - 135

(*cont.*)

R/BASIC, formatting (*cont.*)

- date conversions N4.86 - 88, 4.136 - 138
- datetime conversions N4.88 - 90, 4.138 - 140
- domain-level validation and conversion N3.42
- hexadecimal conversions N4.90 - 91, 4.140 - 141
- justification N4.69 - 70
- masked decimal
 - conversions N4.92 - 93, 4.142 - 144
- masking N4.69 - 70
- numeric to ASCII character string
 - conversions N4.30 - 31
- numeric value of an ASCII character N4.188 - 189
- octal conversions N4.90 - 91, 4.140 - 141
- padding N4.69 - 70
- placing quote marks around expressions N4.168 - 169
- scientific notation
 - conversions N4.93 - 94, 4.144 - 145
- string-for-string global
 - replacements N4.201 - 202
- strings of spaces N4.191 - 192
- time conversions N4.94 - 96, 4.145 - 146
- trimming extra spaces N4.207 - 208
- upper/lower case conversions N4.43
- user-defined conversions N5.12 - 13

R/BASIC, functions

- See R/BASIC, subroutines and functions

R/BASIC, logical expressions

- AND and OR N4.20 - 21
- bit-wise operators N4.22 - 23
- NOT N4.130 - 131

R/BASIC, matrices

- See also R/BASIC, dynamic arrays
- See also R/BASIC, strings
- See also R/BASIC, variables
- assigning dynamic array values to N4.122 - 123

(*cont.*)

R/BASIC, matrices (*cont.*)

- assigning values to N4.14 - 15,
4.119 - 121, 4.123 - 125
- assigning values to dynamic
arrays N4.125 - 126
- assigning values to
records N4.126 - 128
- creating N4.53 - 54
- definition N3.22
- dimensions N3.23, 4.39 - 40, 4.53 - 54
- number of elements N4.102 - 103
- subscripts N3.22 - 23
- vs. dynamic arrays N3.21 - 22

R/BASIC, memory management

- clearing inactive file buffers N4.68 - 69
- flushing N3.24
- reading from buffers N4.175 - 176
- recovering available RAM N4.68 - 69,
4.76
- removing subroutines and functions
at termination N4.63 - 64

R/BASIC, numbers

See also R/BASIC, strings

- absolute value N4.18
- arctangent N4.21 - 22
- arithmetic expressions N3.6 - 7
- arithmetic operators N3.7 - 8, 4.9 - 10
- arithmetic operators (priority
table) N3.9
- calculation limitations N3.4
- cosine N4.43 - 44
- determining if numeric N4.133 - 134
- exponent (base e) N4.62 - 63
- exponentiation N4.167 - 168
- inverse (negative) N4.129 - 130
- logarithm N4.110 - 111
- multivalued arithmetic operators N3.8
- null N3.5 - 6
- precision N3.3
- random number
generator N4.101 - 102, 4.183
- remainder (modulo) N4.128 - 129
- scientific notation N3.3
- sine N4.190 - 191
- square root N4.192 - 193
- substituting symbols for N4.60 - 61
(*cont.*)

R/BASIC, numbers (*cont.*)

- summing arrays N4.198 - 199
- tangent N4.202 - 203
- truncating decimal
fractions N4.107 - 108
- zero N3.5 - 6

R/BASIC, operating system

- accessing files N4.100
- closing files N4.153 - 154
- date N4.47 - 48
- default drive N4.57
- deleting files N4.154 - 155
- executing commands N4.160 - 161,
4.199 - 201
- file errors N3.38
- file names N4.56 - 57
- file size, date, and time N4.54 - 56
- files N3.37 - 40
- internal time N4.203 - 204
- internal time and date N4.204 - 205
- opening files N4.155 - 156
- reading files N4.150 - 152, 4.156 - 157
- sample program N3.39 - 40
- writing to files N4.152 - 153, 4.158

R/BASIC, records

- as dynamic arrays N3.30
- as matrices N3.30
- assigning fields to
variables N4.177 - 178
- assigning matrix
values to N4.126 - 128
- assigning values to
matrices N4.123 - 125
- assigning values to
variables N4.169 - 171, 4.175 - 176
- creating lists of record
keys N4.185 - 189
- creating or updating on
disk N4.210 - 212
- deleting N4.50 - 51
- dynamic arrays N3.16 - 22
- extracting fields from
files N4.213 - 215
- locking and unlocking N4.114 - 117,
4.208 - 210
- sorting lists of record
keys N4.185 - 189

R/BASIC, reports

See also R/BASIC, formatting
 See also R/BASIC, strings
 See also R/BASIC, system subroutines
 column headings N4.37 - 38
 column lengths N4.38 - 39
 page eject N4.160
 page footing N4.71 - 72
 page headings N4.82 - 83
 View window N5.16 - 18

R/BASIC, strings

See also R/BASIC, data entry and output
 See also R/BASIC, dynamic arrays
 See also R/BASIC, numbers
 See also R/BASIC, formatting
 character conversions N4.30 - 31,
 N4.42 - 43
 comparison operators
 (table) N3.10
 comparisons N3.9 - 10
 concatenation N3.11, 4.9 - 10
 constants N3.4
 counting substrings N4.44 - 45
 deleting, inserting, or replacing
 substrings N4.67 - 68
 delimiter positions N4.35 - 37
 determining if alphanumeric N4.19
 determining if numeric N4.133 - 134
 dynamic array extraction and
 assignment N4.10 - 11
 extracting substrings N3.11, 4.65 - 66
 field, value, and subvalue
 positions N4.113 - 114
 global replacements N4.201 - 202
 hex values N3.4
 input conversions N4.83 - 96
 length N4.109 - 110
 length limit N3.4
 null N3.5 - 6
 numeric values of ASCII
 characters N4.188 - 189
 operators N3.11 - 12
 output conversions N4.134 - 146
 pattern matching N4.121 - 122
 quotation marks N3.3 - 4
 (cont.)

R/BASIC, strings (cont.)

repeating N4.195 - 196
 spaces N4.191 - 192
 string extraction and
 assignment N4.3 - 6
 substituting symbols for N4.60 - 61
 substring positions N4.99 - 101,
 4.111 - 113
 upper/lower case conversions N4.43

R/BASIC, subroutines and functions

See also Chapter N5
 See also R/BASIC, system subroutines
 branching to external
 subroutines N4.26 - 27
 branching to internal
 subroutines N4.80
 changing variables in
 subroutines N3.24
 creating external functions N4.74 - 76
 creating external
 subroutines N4.196 - 198
 declaring external functions N3.28,
 4.49
 executing operating system
 commands N4.160 - 161, 4.199 - 201
 executing TCL commands N4.61,
 4.161 - 162
 external functions N3.28 - 29
 external subroutines N3.24
 flushing object code from
 memory N3.24
 internal functions N3.27 - 28
 internal subroutines N3.23 - 24,
 N4.79
 parentheses N3.27
 passing variables by
 reference N3.24 - 25
 passing variables by value N3.25
 recursive subroutines N3.24
 removing from memory at
 termination N4.63 - 64
 returning to calling program from
 subroutines N4.181 - 182
 substituting symbols for
 functions N4.60 - 61
 unconditional branching N4.29

R/BASIC, system subroutines

- Btree indexes N5.6 - 7, 5.10 - 11, 5.15 - 16
- codes and commands N5.7 - 10
- creating and displaying
 - popups N5.22 - 28
- definition N5.1 - 2
- displaying screen
 - messages N5.19 - 22
- Editor N5.32 - 41
- file or record access
 - errors N5.13 - 14
- including external blocks of
 - code N5.2 - 3
- initializing cursors N5.28 - 32
- prompting for data N5.18 - 22
- saving, restoring, and clearing
 - screen N5.47 - 49
- screen borders N5.5
- selecting records N5.28 - 32
- system constants N5.2 - 3
- user-defined conversions N5.12 - 13
- View window N5.16 - 18
- writing to the status line N5.41 - 47

R/BASIC, variables

- See also R/BASIC, branching and flow
- See also R/BASIC, constants
- See also R/BASIC, matrices
- @ variables N3.52 - 59
- @ANS N3.51
- @DICT N3.50
- @ID N3.50 - 51
- @RECORD N3.50 - 51
- AREV.COMMON N3.60 - 65
- assigning SQL rows to K8.42 - 44
- assigning values to N4.11 - 14
- changing values in subroutines N3.24
- clearing N4.31
- clearing COMMON N4.32
- common to any program N4.41 - 42
- common to programs and
 - subroutines N4.39 - 40
- current dictionary file handle N3.50
- current record N3.50 - 51
- current record key N3.50 - 51
- definition N3.2 - 3
- (*cont.*)

R/BASIC, variables (*cont.*)

- naming conventions N3.2
- passing by reference N3.24 - 25
- passing by value N3.25
- substituting symbols for N4.60 - 61
- symbol table N3.1
- symbolic field value N3.51
- transferring data
 - between N4.205 - 206
- user-defined N3.59
- window COMMON
 - variables N3.60 - 65

R/LIST

- See Reports, R/LIST

RAM

- See R/BASIC, memory management

Random numbers

- (R/BASIC) N4.101 - 102, 4.183

READ (R/BASIC) N4.169 - 171

Reading files

- See R/BASIC, files

READNEXT (R/BASIC) N4.171 - 172

READNEXT BY (R/BASIC) N4.172 - 175

READO (R/BASIC) N4.175 - 176

READV (R/BASIC) N4.177 - 178

Record keys

- See also Browse lists
- See also Filters
- See also Indexes
- See also Key columns (SQL)
- See also Query
- See also R/BASIC, cursors
- See also R/BASIC, files
- See also R/BASIC, system subroutines
- See also Reports, R/LIST
- See also Select lists
- ASCII bonded files M2.2
- bonded files M1.13 - 14
- dBASE III bonded files M3.3, 3.5
- defining multipart keys E2.9 - 10
- definition E1.5
- incrementing sequential keys D3.11, F5.7
- lists G1.2 - 7
- multipart F5.6

Record locking

See File and record locking

See R/BASIC, records

See SQL, general

RECORDCOPY (TCL) I3.56 - 59

Records

See also Chapters E1 and E2

See also Editor

See also File and record locking

See also R/BASIC, dynamic arrays

See also R/BASIC, files

See also R/BASIC, records

See also Record keys

See also Rows (SQL)

batch updates F10.1 - 3

copying (TCL) I3.56 - 59

copying in windows D3.9

counting (TCL) I3.13 - 15

defining multipart keys E2.10

definition E1.1

deleting (TCL) I3.19 - 20

deleting in windows D3.9

editing (TCL) I3.25

example E1.2

maximum size E1.6

summing data (TCL) I3.80 - 82

Recursive subroutines

See R/BASIC, subroutines and functions

REDEFINEFILE (TCL) I3.59 - 60

REDUCE (R/BASIC) N5.28 - 32

REFERENCES (SQL) K8.22

Registers F5.51 - 52

Relational indexes F7.7 - 9

Related windows D2.11 - 12, F5.39 - 41

Remainder (modulo) (R/BASIC) N4.128

REMOVE (R/BASIC) N4.178 - 181

RENAMEFILE (TCL) I3.61 - 62

REPEATING (SQL) K8.22

Repeating rows (SQL)

See Columns, multivalued (SQL)

Reports

See Query

See R/BASIC, formatting

See R/BASIC, reports

Reports, EasyWriter

See also Chapter H5

See also Reports, R/LIST

accessing H5.2 - 3

accessing (TCL) I3.27

calling as a subroutine H5.15

editing default settings H5.8, 5.13, 5.15 - 16

grouping records H5.12 - 13

headings and footings H5.13

help H5.4

importing R/LIST commands H5.16

popups H5.4 - 5

reformatting fields H5.12

retrieving from library H5.7

routing output to printer or screen H5.14

saving H5.3 - 4

selecting records H5.7 - 11, 5.14

sorting records H5.11

totals and averages H5.12

Reports, Forms

See also Chapter H2

accessing Paint (TCL) I3.51 - 52

converting windows to H2.3

deleting H2.5

displaying field names H2.5

editing existing H2.1

editing prompt characteristics H2.6

executing (TCL) I3.31 - 32

footing H2.9

formatting H2.7 - 9

heading H2.9

hiding prompts H2.5

Painting H2.1 - 5

printing H2.10

routing to printer or screen H2.9

saving H2.4

testing H2.10

virtual space H2.4

Reports, labels

See also Chapter H3

customizing field attributes H1.2 - 3

executing (TCL) I3.34 - 35

Reports, Merge

See also Chapter H6

conditional scripts H6.12 - 13

creating a default Merge

template H6.23

creating Merge templates H6.2 - 7

debugging H6.17 - 18

definition H6.1 - 2

executing H6.20 - 21

executing (TCL) I3.49 - 50

field flags H6.8 - 9

footers H6.7

forcing page eject H6.10

formatting characters H6.13 - 14

formatting fields H6.14 - 17, 6.23 - 24

formatting multivalued

blocks H6.10 - 12

formatting paragraphs (table) H6.10

formatting paragraphs,

default H6.5 - 6

formatting paragraphs, flags H6.9 - 10

headers H6.6

importing scripts H6.23

indenting paragraphs H6.4 - 5

margins H6.4

naming Merge templates H6.3

page dimensions H6.5

printers H6.6, 6.22 - 23

quick reference chart H6.24 - 27

reserved words H6.9

routing output to printer, screen, or

file H6.19 - 21

scripts (text) H6.7 - 8

special text H6.22

testing H6.17 - 19

widows and orphans H6.5

Reports, R/BASIC

See R/BASIC, reports

Reports, R/LIST

See also Chapters H4, J1, and J2

See also Reports, EasyWriter

accessing from windows D2.13

averages J2.7

capturing R/BASIC source code J1.2

case sensitivity J1.1

column headers H1.3, J2.10

(*cont.*)

Reports, R/LIST (*cont.*)

command format J2.1

converting formulas to symbolic
fields J1.5 - 6

customizing field attributes H1.2 - 3

definition J1.1

dictionaries accessed J1.1, 2.20

displaying in View window H1.4 - 7

double-spacing J2.4

editing output defaults J1.5

executing (TCL) I3.35 - 37, 3.64,
3.75 - 78

exploding sorts J2.9 - 10

field display length H1.3, J2.15

field justification H1.3, J2.15

footers J2.11 - 12

forcing page eject J2.8

format keywords J2.4 - 5

formulas J1.3 - 5

grand totals J2.12 - 13

grouping sorted data J2.7 - 9

headers J2.13 - 14

listing record keys only J2.18 - 19

output format conversions J2.18

printing J2.4

select lists J1.6 - 14

selecting records J2.21 - 26

selecting records based on

multivalued fields J2.16 - 17

sorting records J2.9 - 11

status line J1.9

suppressing column headings J2.4

suppressing detail J2.4

suppressing headers J2.4

suppressing page breaks J2.4

suppressing record keys J2.4

synonyms for commands and

keywords J1.3

syntax J2.1 - 3

throwaway words J2.6

totals J2.19 - 20

View window H1.4 - 7

Reports, SQL

See SQL, queries

Requirements

See System requirements

RESET (TCL) I3.62

Resolved select lists
 See R/BASIC, cursors
 See Select lists
 Results tables (SQL)
 See Tables (SQL)
 RETURN (TCL) I3.63
 RETURN/RETURN TO
 (R/BASIC) N4.181 - 182
 Rightdex indexes F7.10 - 12
 RLIST (TCL) I3.64
 See also Reports, R/LIST
 RND (R/BASIC) N4.183
 ROLLBACK (SQL) K8.46 - 47
 ROLLOUT (TCL) I3.65
 ROS files E1.3, 2.14
 Rows (SQL)
 See also Columns (SQL)
 See also Columns, multivalued (SQL)
 See also R/BASIC, records
 See also Records
 adding K8.44 - 45
 assigning to R/BASIC
 variables K8.42 - 44
 counting K4.3, 8.13
 definition K2.2
 deleting K5.6 - 7, 8.39 - 41
 evaluating K8.26 - 28
 grouping K4.9 - 10, 8.16
 inserting K5.1 - 4
 inserting values K8.17
 selecting all K8.10
 selecting any/some K8.11, 8.23
 selecting distinct K8.14
 selecting groups of K8.16 - 17
 selecting null/not null values K8.20
 selecting ranges K8.11
 selecting using wildcard
 characters K8.18
 sorting K4.8 - 9, 8.21
 suppressing display K8.24
 testing for inclusion of values K8.17
 RUN (TCL) I3.65 - 66
 RUNMENU (TCL) I3.66 - 67

S

SAVELIST (TCL) I3.67
 Scientific notation
 See R/BASIC, formatting
 See R/BASIC, numbers
 Screen
 See also R/BASIC, data entry and
 output
 See also R/BASIC, system
 subroutines
 See also Video attributes
 displaying current settings
 (TCL) I3.70
 editing dimensions (TCL) I3.70
 saving, restoring, and clearing
 (R/BASIC) N5.47 - 49
 SCRIBE (R/BASIC) N5.32 - 41
 Searches
 See Indexes
 Security
 See also File and record locking
 creating or changing passwords
 (TCL) I3.52
 disabling keys F5.43
 encrypting templates (TCL) I3.25 - 26
 locking records during transaction
 processing E2.26 - 27
 logto passwords F3.11
 menu displayed at logon F6.3
 passwords F3.4 - 6
 restricting access to accounts F3.7,
 3.10 - 11
 restricting access to windows F3.7
 restriction levels F3.6
 user names and tables (SQL) K2.6,
 8.25
 users F3.12 - 13
 windows F5.41 - 42
 SELECT (R/BASIC) N4.184 - 188
 SELECT (SQL) K8.47 - 53, 8.55
 SELECT (TCL) I3.68 - 69

Select lists

See also Browse lists
 See also Query
 See also R/BASIC, cursors
 See also R/BASIC, files
 See also R/BASIC, system subroutines
 See also Record keys
 See also Reports, R/LIST
 clearing (TCL) I3.8
 creating (TCL) I3.68 - 69, 3.78 - 80
 creating from data (TCL) I3.46 - 47
 creating with Easywriter H5.7 - 11, 5.14
 creating with the Filter key
 [Ctrl-F10] G2.1 - 3
 definition G1.2 - 3
 deleting (TCL) I3.21
 executing filters (TCL) I3.30
 inline formulas J1.9 - 10
 latent vs. resolved J1.11 - 14
 passing to windows as browse
 lists G1.6 - 7
 retrieving G2.2
 retrieving (TCL) I3.32
 saving G2.2
 saving (TCL) I3.67
 status line G2.3
 stored in Query table G2.2
 SEQ (R/BASIC) N4.188 - 189
 Sequential record keys D3.11, F5.7
 SERIAL (R/BASIC) N4.189 - 190
 Serial number (R/BASIC) N4.189 - 190
Servers
 See Database servers
 See File servers
SET (SQL) K8.23
SET TRANSACTION (SQL) K8.55 - 56
SET-CRT (TCL) I3.70
SET-LPTR (TCL) I3.71
SET-OPTIONS (TCL) I3.72
SETFILE (TCL) I3.72 - 74
SETVOLUME (TCL) I3.74 - 75
SIN (R/BASIC) N4.190 - 191
Sine (R/BASIC) N4.190 - 191

Softkeys

See also Macro keys
 creating in Paint F5.39
 definition D2.18
 in Editor D4.10 - 11
 in Paint F4.23
 at TCL I1.2 - 3
SOME (SQL) K8.23
SORT (TCL) I3.75 - 78
SORT ONLY (R/LIST)
 See ONLY (R/LIST)
 Sorting records J2.9 - 11
 Source code (R/BASIC) N3.1
SPACE (R/BASIC) N4.191 - 192
 Spaces, strings of
 (R/BASIC) N4.191 - 192
SQL, general
 accessing files K2.9
 accessing SQL window K1.1
 accessing SQL window (TCL) I3.78
 accessing the master dictionary K6.7
 Advanced Revelation
 extensions to K8.1 - 3
 comparison with Advanced
 Revelation K2.7 - 8
 data types K8.5
 database terminology K2.8
 definition K2.1 - 2
 implicit table and record
 locking K6.5 - 6, 8.12, 8.55 - 56,
 L3.8 - 13
 listing dictionary definitions E3.8
 maximum number of locks K6.5
 multivalued columns K2.9 - 10
 naming columns, tables, and
 views K2.5 - 6, 2.8 - 9
 notation (table) K8.6
 null handling K2.10 - 11
 operators (table) K8.6
 reserved words (list) K8.4
 security K2.6, 8.25
 SQL Assistant K1.2
 statements K2.3 - 5

SQL, creating and altering tables and creating views

See also Chapter K3

accessing views K3.9

adding columns K8.10, 8.31 - 33

adding rows to tables K8.44 - 45

altering table definitions K3.7, 8.19, 8.31 - 33

column default values K3.2 - 3

creating tables K8.34 - 36

creating views K3.8 - 9, 8.11, 8.36 - 37

data types K3.2

date and time K8.13

default column values K3.2 - 3, 8.14

defining columns K3.1

defining multiple columns K3.6 - 7

defining unique columns K8.24

deleting columns K8.15, 8.31 - 33

deleting rows K5.6 - 7, 8.39

deleting tables K5.8, 8.41

deleting views K5.8, 8.41 - 42

documenting references K8.22

foreign keys K3.5, 8.22

inserting rows K5.1 - 4

inserting values in columns K8.17, 8.25

key columns K3.4 - 5, 8.22

keyword order K3.1

multivalued columns (repeating rows) K3.6, 8.22

multivalued key columns K8.17 - 18

null values K8.20

requiring data K3.3 - 4

testing UPDATES and INSERTs for conformity K8.29

updating columns K5.4 - 6, 8.23, 8.57 - 58

user name column K8.25

views (definition) K3.8

volumes K3.6, 8.20

SQL, programming (embedded SQL)

See also Chapter K7

activating cursors K8.46

assigning rows to R/BASIC variables K8.42 - 44

branching K7.9 - 11, 8.16, 8.19, 8.26
(*cont.*)

SQL, programming (embedded SQL) (*cont.*)

branching on errors K8.23

capturing source code K7.13

closing cursors K8.33

compiled vs. interactive SQL K7.1

compiling programs K7.11 - 13

continuing execution after error condition K8.13

creating queries K8.51 - 8.55

cursors K7.6 - 9, 8.38

definition K7.2

deleting rows K8.40 - 41

error indicators K7.9 - 11

positioning cursor K8.42 - 44

referencing queries K8.14

referencing variables assigned to cursors K8.14

referencing variables storing rows K8.17

SQL Module Language K7.1

statements K7.2 - 3

subroutines K8.15

syntax K7.3 - 4

transaction processing K7.13

updating columns K8.58 - 60

variables K7.4 - 5

SQL, queries

averaging columns K4.3, 8.11

calculating with columns K4.4

combining tables (unions) K4.10 - 11, 8.24 - 25

conditional selects K4.4 - 8

counting rows K4.3, 8.13

creating K8.47 - 51

evaluating rows K8.26 - 28

existence queries K4.14

format K4.1

grouping rows K4.9 - 10, 8.16 - 17

joining tables K4.11 - 12

nesting queries K4.13 - 14

printing tables K8.24

selecting all K8.10

selecting all columns K4.2

selecting any/some K8.11, 8.23

selecting distinct rows K8.14
(*cont.*)

SQL, queries (*cont.*)

- selecting maximum and minimum values K4.4, 8.18 - 19
- selecting null/not null values K8.20
- selecting ranges K8.11
- selecting rows using wildcard characters K8.18
- selecting values from multivalued columns (repeating rows) K4.15
- sorting rows K4.8 - 9, 8.21
- subqueries K4.13 - 14
- suppressing row display K8.24
- table names, temporary K4.12
- testing for existence K8.15
- testing for inclusion K8.17
- totaling columns K4.3, 8.23
- unions K4.10 - 11, 8.24 - 25

SQL, table and record locking

- concurrency controls K8.55 - 56
- consistency levels K8.12, 8.55 - 56

SQL, transaction processing

- See also Chapter K6
- See also Transaction processing
- beginning K6.3, 8.33
- commit log volume K6.1
- definition K2.6 - 7
- embedded SQL K7.13
- executing K8.34
- locking K6.2 - 6, 8.12, 8.55 - 56, L3.8 - 13
- queuing committed transactions K6.1 - 2
- terminating execution K8.46 - 47
- transaction file volume K6.3

SQLERROR (SQL) K8.23**SQRT (R/BASIC) N4.192 - 193****Square roots (R/BASIC) N4.192 - 193****SSELECT (TCL) I3.78 - 80****Starting Advanced Revelation**

- See Logon

Statements

- See R/BASIC, compiler
- See SQL, general

Station number, networks L2.10

- See also Networks

STATUP (R/BASIC) N5.41 - 47**Status line**

- in Editor D3.8
- in filters G2.3
- in Help D2.3 - 4
- in Paint F4.7 - 8
- in Pan mode D2.14 - 15
- in popups D2.18
- in R/LIST reports J1.9
- in Resize mode D2.16
- in windows D2.9 - 10
- layout D2.5 - 6
- percentage done and status bar J1.9
- writing to (R/BASIC) N5.41 - 47

STATUS() (R/BASIC) N3.35, 4.193 - 194

- See also R/BASIC, errors

- See also R/BASIC, files

STOP (R/BASIC) N4.194 - 195**Stop lists (Cross Reference**

- indexes) F7.4 - 5

Stopping program execution

- See R/BASIC, branching and flow

STR (R/BASIC) N4.195 - 196**String space**

- See R/BASIC, memory management

Strings

- See R/BASIC, branching and flow
- See R/BASIC, constants
- See R/BASIC, data entry and output
- See R/BASIC, dynamic arrays
- See R/BASIC, formatting
- See R/BASIC, strings

Subqueries (SQL) K4.13 - 14**SUBROUTINE (R/BASIC) N4.196 - 198****Subroutines**

- See R/BASIC, branching and flow
- See R/BASIC, subroutines and functions
- See R/BASIC, system subroutines
- See SQL, programming

Subscripts

- See R/BASIC, dynamic arrays
- See R/BASIC, matrices

Substrings

- See R/BASIC, strings

Subvalue marks N3.16 - 17

SUM (R/BASIC) N4.198 - 199
 SUM (SQL) K8.23
 SUM (TCL) I3.80 - 82
 Summing dynamic arrays
 (R/BASIC) N4.198 - 199
 SUPP (R/LIST) J2.4
 SUPPRESS DISPLAY (SQL) K8.24
 Suppressing detail
 See Reports, R/LIST
 Suppressing error messages (TCL)
 See individual TCL commands
 Suppressing page eject
 See Page eject
 SUSPEND (R/BASIC) N4.199 - 201
 SUSPEND (TCL) I3.65, 3.82 - 83
 See also Operating system
 See also PC (TCL)
 SWAP (R/BASIC) N4.201 - 202
 Symbol table (R/BASIC) N3.1
 Symbolic fields
 See Fields
 See R/BASIC, fields
 Synonyms
 See Fields, synonyms
 See Files, synonyms
 See Reports, R/LIST
 See TCL
 SYSPROG (System Programmer)
 account F3.4, 3.9
 System configuration
 See also Environment
 displaying current status
 (TCL) I3.89 - 90
 displaying defaults (TCL) I3.72
 editing defaults (TCL) I3.72
 expanded memory B5.6 - 8
 terminal emulation (TCL) I3.84 - 86
 System files
 affected by installation B2.9 - 10
 application (list) E2.28
 global (list) E2.29
 user and account
 information E2.29 - 30
 System, re-initializing (TCL) I3.62
 System requirements
 expanded memory B5.1
 installation B1.1

System subroutines
 See R/BASIC, system subroutines
 System variables
 See R/BASIC, variables

T

Tables (SQL)
 See also Files
 See also R/BASIC, files
 See also Views (SQL)
 assigning to volumes K8.20
 combining (unions) K4.10 - 11,
 8.24 - 25
 creating K8.34 - 36
 definition K2.1
 deleting K5.8, 8.41
 editing definitions K3.7
 joining K4.11 - 12
 naming K2.5
 printing K8.24
 temporary names K4.12
 updating with cursors K7.8
 TAN (R/BASIC) N4.202 - 203
 Tangent (R/BASIC) N4.202 - 203
 TCL (The Command Level)
 accessing I1.2
 accessing (R/BASIC) N4.61 - 62,
 N4.161 - 162
 accessing from windows D2.11
 accessing command windows I1.5
 adding commands E2.16
 adding R/BASIC programs to VOC
 file I3.48 - 49
 batch processing commands I1.5,
 3.2 - 4
 creating command synonyms E2.17
 definition I1.1
 deleting commands E2.17, I3.88 - 89
 deleting R/BASIC programs in VOC
 file I3.88 - 89
 linking multiple commands I1.3
 retrieving previously-executed
 commands I1.2
 returning to previous level I3.63
 saving commands I1.4
 softkeys I1.2 - 3

TERM(INAL) (TCL) I3.84 - 86
Terminal emulation
 downloading data from the host
 (TCL) I3.86
 settings (TCL) I3.84 - 85
Terminating cursors
 See R/BASIC, cursors
Text editing
 See Editor
The Command Level
 See TCL
Time (SQL) K8.13
TIME (TCL) I3.86 - 87
TIME() (R/BASIC) N4.203 - 204
TIMEDATE() (R/BASIC) N4.204 - 205
TO PRINTER (SQL) K8.24
TOTAL (R/LIST) J2.19 - 20
TRANSACTION (TCL) I3.87 - 88
Transaction processing
 See also File and record locking
 See also SQL, general
 See also SQL, transaction processing
 commitlog report (TCL) I3.9
 commitlog E2.24 - 26
 default volume E2.23 - 24
 definition E2.18
 displaying locking information
 (TCL) I3.42 - 43
 processing sequence E2.20 - 21
 record locking E2.26 - 27
 status report E2.27
 system failure E2.21 - 23
 transaction control (TCL) I3.11 - 13,
 3.87 - 88
 transaction control E2.19 - 20
 unlocking transactions at
 commit L3.7
TRANSFER (R/BASIC) N4.205 - 206
Translate
 See R/BASIC, files
Trigonometry
 See R/BASIC, numbers
TRIM, TRIMB, TRIMF
 (R/BASIC) N4.207 - 208
Trimming extra spaces
 (R/BASIC) N4.207 - 208

Tutorial
 See also Chapter C1
 accessing (TCL) I3.88
 active keys C1.7
 clearing messages C1.5
 help C1.5
 keystroke playback speed C1.6
 keystrokes C1.7
 starting C1.1 - 3
 stopping C1.3 - 5
TUTORIAL (TCL) I3.88

U
UNION (SQL) K8.24 - 25
UNIQUE (SQL) K8.24
UNLOCK (R/BASIC) N4.208 - 210
UNMAKEVOC (TCL) I3.88 - 89
UPDATE (SQL) K7.8, 8.57 - 60
Updates to Advanced Revelation
 See Installation
Updating files
 See R/BASIC, files
Upgrades to Advanced Revelation
 See Installation
Upper/lower case conversions
 Editor D4.12
 R/BASIC N4.43
USER (SQL) K8.25
User-defined conversions
 (R/BASIC) N5.233 - 234
Users
 See also Chapter F3
 See also Environments
 creating F3.12
 creating (TCL) I3.47
 definition F3.3
 deleting F3.12
 deleting (TCL) I3.20 - 21
 displaying current status
 (TCL) I3.89 - 90
 increasing number on network
 (TCL) I3.7
 security F3.12 - 13
USING (R/LIST) J2.20

V

- Validation patterns F5.19 - 27
- Value marks N3.16 - 17
- VALUES (SQL) K8.25
- Variable passing
 - See R/BASIC, branching and flow
 - See R/BASIC, subroutines and functions
- Variables (R/BASIC)
 - See R/BASIC, constants
 - See R/BASIC, dynamic arrays
 - See R/BASIC, matrices
 - See R/BASIC, subroutines and functions
 - See R/BASIC, variables
- Variables (SQL)
 - assigned to cursors K8.14
 - passing data to and from statements K7.4 - 5
 - referencing queries K8.14
- Verifile (data validation) F5.22 - 23
- Video attributes
 - See also Screen
 - definition F3.15
 - editing F3.16
 - editing in Paint F4.19, 5.41
- VIDEO.RW (R/BASIC) N5.47 - 49
- View window H1.4 - 7
 - See also R/BASIC, system subroutines
- Views (SQL)
 - See also Tables (SQL)
 - accessing K3.9
 - creating K3.8 - 9, 8.36 - 37
 - definition K3.8
 - deleting K5.8, 8.41 - 42
 - naming K2.5
 - testing UPDATES and INSERTs for conformity K8.29
- Virtual space
 - definition D2.14
 - in report forms H2.4
 - size limitations F4.5
- VOC file
 - adding R/BASIC programs (TCL) I3.48 - 49
 - deleting R/BASIC programs (TCL) I3.88 - 89

Volumes

- See also Chapter E4
- See also Files
 - accessing (TCL) I3.1, 3.22
 - accessing via LOGTO (TCL) I3.43 - 44
 - attaching E4.2
 - bonded (foreign) files E4.4, M1.5 - 6
 - creating E4.4
 - DATAVOL E4.3
 - default E4.2 - 3, 4.5 - 6
 - definition E4.1
 - detaching E4.3
 - directories E4.6
 - editing defaults E4.5 - 6
 - initializing (TCL) I3.50 - 51
 - listing E4.5
 - listing attached (TCL) I3.42
 - listing files (TCL) I3.41 - 42
 - naming E4.4
 - naming (TCL) I3.74 - 75
 - renaming (TCL) I3.50 - 51
 - REVBOT E4.3
 - SQL commit log volume K6.1
 - SQL tables K3.6, 8.20
 - SQL transaction file volume K6.3
 - TRANSACT E2.23 - 24
 - transaction processing E2.23 - 24

W

- WHENEVER (SQL) K8.26
- WHERE (SQL) K8.26 - 28
- WHO (TCL) I3.89 - 90
- Widows (Reports, Merge) H6.5
- WINDOW (TCL) I3.90 - 91
- Windows
 - See also Data entry and editing
 - See also Paint, windows
 - See also R/BASIC, system subroutines
 - See also Reports, R/LIST
 - accessing D2.7
 - accessing (TCL) I3.90 - 91
 - accessing application windows (TCL) I3.2
 - accessing command windows (TCL) I3.15
 - (cont.)

Windows (cont.)

- accessing from TCL I1.5
- accessing Main Menu D2.10
- accessing R/LIST D2.13
- accessing related D2.11 - 12
- accessing TCL D2.11
- application (definition) D2.7
- collector (definition) D2.7
- converting to report forms H2.3
- copying fields and records D3.9
- cursor movement D3.1 - 5
- definition D2.7 - 8
- deleting records D3.9
- encrypting (TCL) I3.25
- highlighting editable prompts D3.6
- moving on the screen D2.13
- panning through virtual space D2.14 - 15, 2.17
- Query D2.12
- refreshing D2.8
- Reports, Merge H6.5
- resizing D2.15 - 17
- saving data D2.8
- softkeys D2.18
- status line D2.9 - 10
- zooming D3.10

WITH (R/LIST) J2.21 - 26

WITH CHECK OPTION (SQL) K8.29

WORK (SQL) K8.29

Workspace

- See R/BASIC, memory management

Workstation number, networks L2.1

- See also Networks

WRITE (R/BASIC) N4.210 - 212

WRITEV (R/BASIC) N4.212 - 213

Writing files

- See R/BASIC, files

Z**Zero**

- See R/BASIC, numbers

Zoom D3.10

ZUMZUM I3.52

X

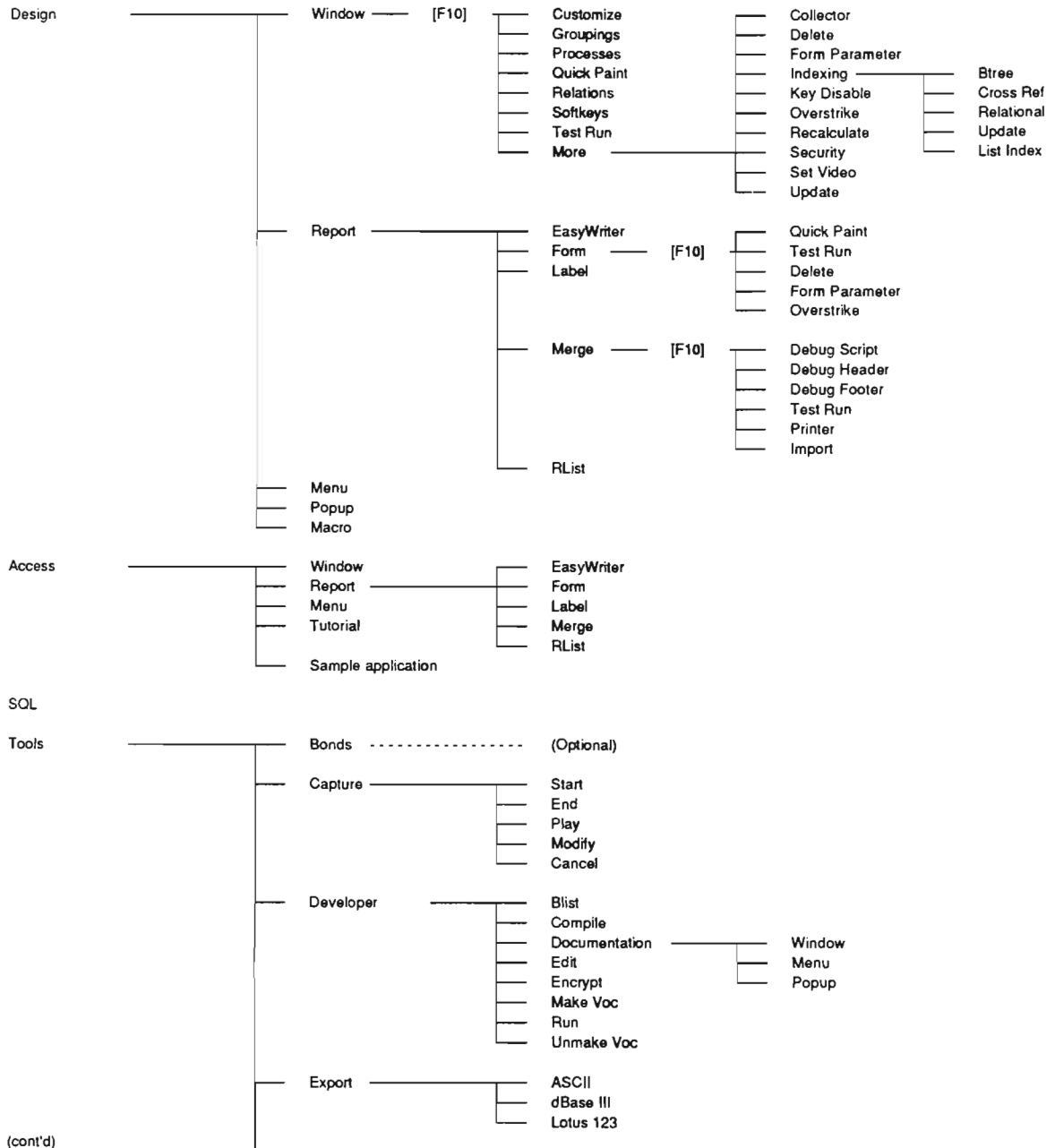
XLATE (R/BASIC) N4.213 - 215

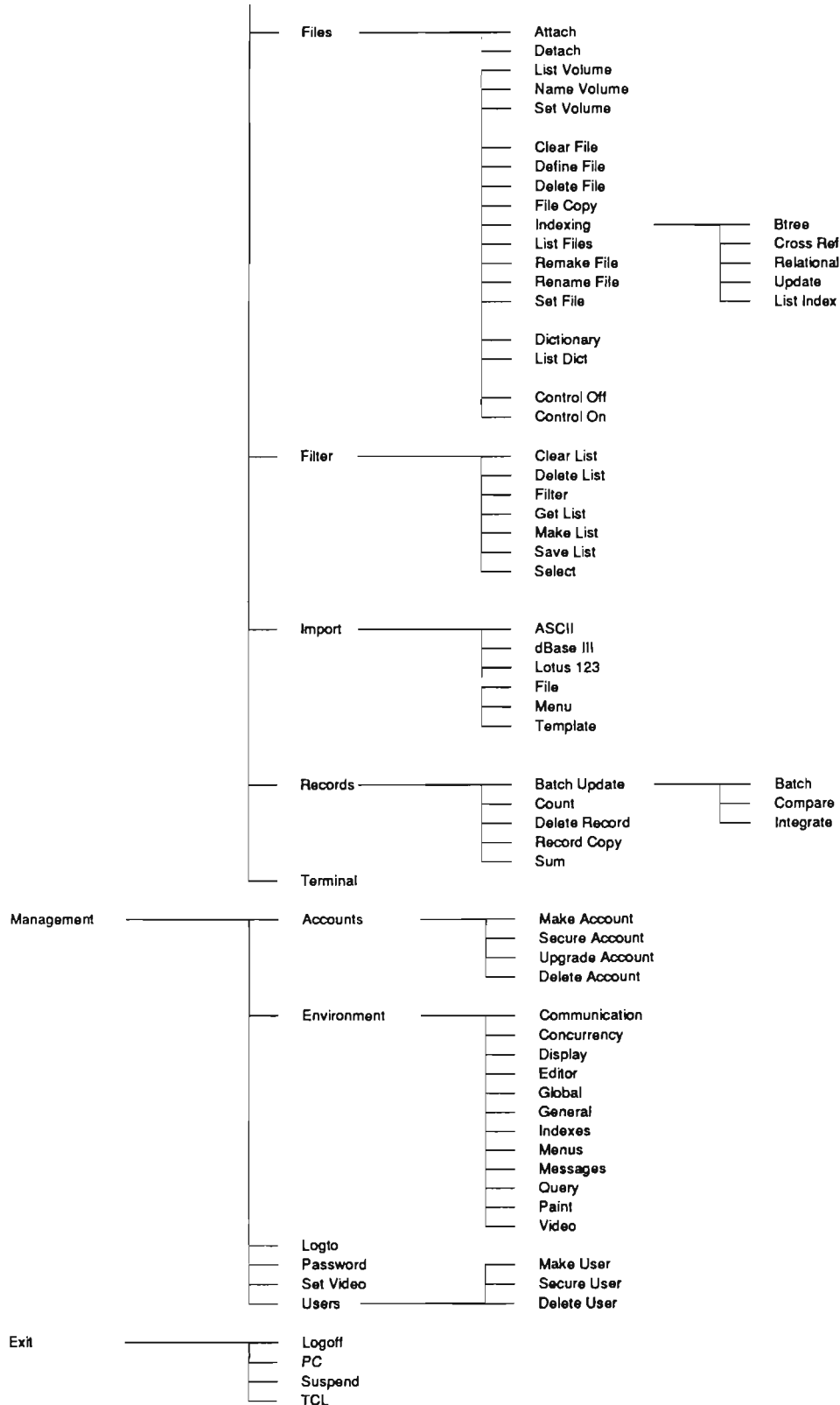
XYPHIS I3.52

Advanced Revelation Menus



Advanced Revelation Menus







MEMSPACE - FUNCTION - RETURNS AVAILABLE MEMORY

IF ARG IS BIGGER THAN WHATS AVAILABLE.

DECLARE FUNCTION MEMSPACE

X = MEMSPACE(9999999)

PRINT X WILL PRINT AVAILABLE MEMORY.





