ADVANCED REVELATION®

R / B A S I C

Revelation Technologies

Advanced Revelation®

Version 3.0

R/BASIC

SER# 35474

COPYRIGHT

TRADEMARK NOTICE

SOFTWARE COPYRIGHT NOTICE

Advanced Revelation Version 3.0
October 1992

Table of contents

Introduction to R/BASIC

1. About 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 rows 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.

An important note about the words "file", "record", and "field"

Throughout Advanced Revelation you will see references to "tables", "rows", and "columns" when we speak about storing or manipulating data. In this R/BASIC book, however, you will note that we use the alternative terms "files", "records", and "fields". As used in this book, the terms are completely equivalent—a files is a table is a file, a record is a row, and a field is a column.

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 that 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 provides the means for building what in essence is a structure for the input of data into tables. 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 table 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 would be an invoicing system in which users enter information about orders, after which 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 that 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 tables, format output, calculate quantities, and more. R/BASIC is also the basis for virtually all the commands used every day in Advanced Revelation — COPYROW, SUMROWS, 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 about 34K of source code, or 64K in Advanced Revelation for OS/2). 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 be called by other programs, or by code and command processes. Stand-alone programs are kept as entries in an Advanced Revelation table. No distinction is made between data and program tables, but it is not a good practice to keep R/BASIC programs in data tables or in the VOC table.

Each application that you create in Advanced Revelation has two tables already created that are intended for R/BASIC programs. When you write programs, you can

put the source code into the table called SOURCE. When the programs are compiled, they will then be automatically be put into the table called OBJECT. However, you can put your programs wherever you like.

The process of actually creating a program requires several steps:

- Use the option **Define-Program-Edit** from the Development menu 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 table as the source code, but it has a dollar sign ($) in front of the name. If you've put your source code into the table called SOURCE, the object code will automatically be put into the table called OBJECT.

At this point, the TCL command RUN will execute the stand-alone program, and return to the Command window when finished. You also have the option of putting an entry in the VOC table for this program by using the SETPROGRAM command. Users can then just type in the program name at the Command window to execute it.

For example, these steps (executed from the Command window) illustrate a typical scenario:

```
EDIT SOURCE INVOICE_PGM
```

Create or edit the source code for a program called INVOICE_PGM stored in the SOURCE table.

```
COMPILEBASIC SOURCE INVOICE_PGM
```

Translate the source code into Advanced Revelation object code. The result is an item in the OBJECT table called $INVOICE_PGM.

```
RUN OBJECT INVOICE_PGM
```

Execute the program.

```
SETPROGRAM OBJECT INVOICE_PGM
```

Put an entry for this program in the VOC table. From this point, the program can be run by typing INVOICE_PGM at the Command window.

Note The RUN and SETPROGRAM commands pay attention only to the object code (the $ form of the program), so that the source code can be removed from the table. 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 column in the same dictionary item. For information about creating formulas, see the chapter "Building formulas".

Listing R/BASIC programs

A command is provided in Advanced Revelation to create special formatted listings of R/BASIC programs. This command, LISTBASIC, 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 rows in an Advanced Revelation table 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 columns and stand-alone programs can access data by way of dictionary column 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 "Using Advanced Revelation on a network" in the *User's Guide*, 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 your need for sophisticated programming increases, you will find it useful to investigate the full range of tools available in the language.

2. R/BASIC command summary

R/BASIC command summary

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".

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) is 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.
GET-BREAK( ) GETECHO( ) GETPRINTER( ) GETPROMPT( )	Returns the value of the current break key, echo, printer, and prompt character settings.
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 (table) 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 dynamic 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.

Note 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.
OCONV( )	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 *n* characters starting at column *m* 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 returns 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 *prog.name* 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 window 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.

3. Programming with R/BASIC

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 table. The source code for each program can be up to about 34K long (64K in Advanced Revelation for OS/2). Any number of programs can be stored in a table.

When you compile a program, Advanced Revelation generates two additional records and stores them in the table that contains the source code. The object code record name is the same as the program name, but is preceded with a dollar sign ($). For example, the object code version of the program POST_INVOICES is called $POST_INVOICES.

The compiler also generates a table of variables (a "symbol table") for the program. The symbol table name is the same as the program name, but is preceded with an asterisk (*). The R/BASIC debugger uses the symbol table primarily 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 statements, 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 the accessing, processing, and storing of data throughout a program.

```
Letter [A-Z] [0-9] [._$@%]
```

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, or the following special characters:._$@%. Characters from foreign alphabets are allowed as long as they appear in the definition (for that language set) of upper- and lower-case letters. Spaces and minus signs (-) within a name are not allowed.

Variable names can appear in either upper or lower case letters, but 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 CHECK_BAL may be assigned the value "Checkbook Balance" at the beginning of a program and may end the program with the value $110.95.

Note 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
CHECK_BAL ="Checkbook Balance "
CHECK_BAL = CHECK_BAL : DEPOSIT - CHECKS
PRINT CHECK_BAL
```

The final contents of the variable CHECK_BAL is printed as Checkbook Balance followed by the calculated amount.

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 positive or negative number, including scientific notation, that represents a quantity. 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 exponential digits, in that order. Scientific notation may be used at input or whenever numeric constants are designated.

Caution! Numbers smaller than 1E-15 (.000000000000001) now display as 0 (zero).

Numeric constant limits

R/BASIC offers a choice of either 15 or 18 decimal digits of precision. System default is 15 digits. The system always operates with up to 18 digits, but rounds the three least-significant digits, when precision is specified at 15 digits.

Each degree of decimal precision has its benefits. The 15-digit approach, with its internal rounding, is better for business applications, while the 18-digit format is better for scientific applications.

Note Converting numeric values from one degree of precision to another, or from numeric to string (and vice versa), always introduces possibilities for corruption of numeric values. Use caution when undertaking such changes.

Any value greater than 18 digits is truncated, not rounded.

If you use scientific notation, you can enter numbers up to the calculation limits listed below.

Calculation limitations

In Advanced Revelation for DOS, R/BASIC supports calculations to these sizes:

15-digit	18-digit
Binary floating point has an 18-decimal digit mantissa, with 4-digit exponent	Binary floating point is converted to an 18-digit mantissa, which is then rounded to 15 digits
Trailing zeroes in the mantissa are truncated; the range of possible values is approximately 3.4 X 10-4932 — 1.2 X 104932	Trailing zeroes in the mantissa are truncated; the range of possible values is 1.0 X 10-15 — 1.2 X 104932
	Numbers less than or equal to 1X10-15 are rounded to zero

In Advanced Revelation for OS/2, the limits are:

Positive: +10**-307 to +10**+308
Negative: -10**-307 to -10**+308

Character string constants

A string constant is a sequence of characters that may include letters, numbers, dollar signs, or other keyboard characters. It may be up to 254 characters.

String constants must be enclosed in single or double quotation marks ('aaa' or "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 is denoted by two quotation marks with nothing between them ("") or ('').

Note 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.

Hex constants

To include non-printing (or other) characters using their hex value, enclose the hex representation of the characters with backslashes (\).

```
\0D0A\
```

For example:

```
\0D0A\
\FE\
```

The first example is the hex constant that represents a carriage return-line feed combination. The second example represents 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"

Null and zero

Null and 0 (zero) are given special consideration in R/BASIC.

- In arithmetic statements, null is treated as a numeric zero.
- Outside arithmetic statements, a null has 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 0 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.

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 +++, — (3 minus signs), ***, 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 = 1²1²1²1
B = 2²2²2²2²22
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 on the next page 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.

For purposes of comparison, any value less than 0.00005 is considered to be zero. For example, the expression

```
IF 0.0000499 = 0.0000123
```

will evaluate true.

The four digits of precision are *decimal* places, not simply the four most significant digits of a number.

Comparing full values

You can compare values having more than 4 decimal places, by using a special set of comparison operators:

Operator	Meaning
_EQX	Equal, to full precision
_NEX	Not equal, to full precision
_LTX	Less than, to full precision
_GTX	Greater than, to full precision
_LEX	Less than or equal, to full precision
_GEX	Greater than or equal, to full precision

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
_EQC	Equal, case-insensitive
_NEC	Not equal, case-insensitive
_LTC	Less than, case-insensitive
_GTC	Greater than, case-insensitive
_LEC	Less than or equal, case-insensitive
_GEC	Greater than or equal, case-insensitive
_EQX	Equal, to full precision
_NEX	Not equal, to full precision
_LTX	Less than, to full precision
_GTX	Greater than, to full precision
_LEX	Less than or equal, to full precision
_GEX	Greater than or equal, to full precision

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 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 system time and date concatenated.

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, or the following: ._$@%. They may be of any length.

Colons

A colon following a numeric label is optional. A colon following an alphanumeric label is mandatory.

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.

Note Although Advanced Revelation accepts statement identifier labels with the same name as variables in a program, it is not a recommended programming practice.

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

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."

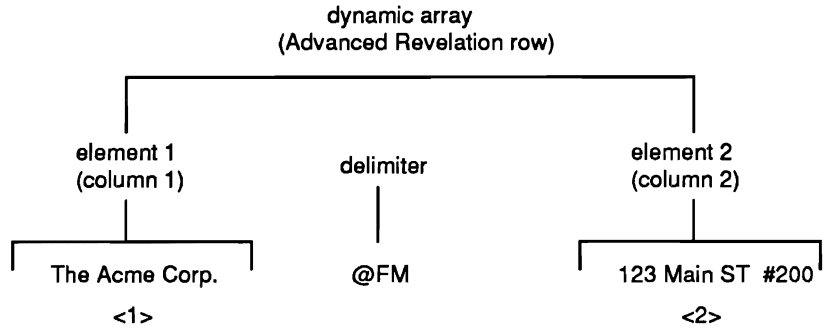
Correct use of conditional branching

```
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 that 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

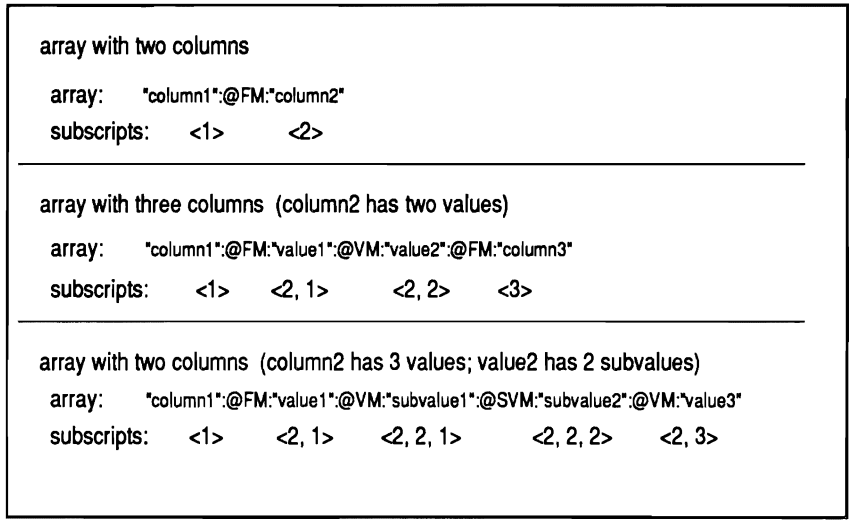
A record is a dynamic array. The elements of a record are defined by special delimiters. Fields, values and subvalues are defined by field marks (ASCII character 254), value marks (ASCII character 253), and subvalue marks (ASCII character 252). A dynamic array can have three dimensions using these delimiters. R/BASIC provides the system variables @FM and @VM for use as a field mark and a value mark, respectively. 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 table
READV	Assigns a field in a record to a variable
WRITEV	Writes a variable to a field

Note 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:



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_TITLES = 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.

```
STRING = CUST_REC<6>
LOCATE "MRS" IN STRING 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 multivalued 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 matrices

```
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
```

(The following code is created and compiled separately.)

```
SUBROUTINE SUB
* subroutine
COMMON X,Y,Z
PRINT X:Y:Z
RETURN
```

A second example:

```
* main program
A = 1
B = 2
C = 3
CALL SUB(A,B,C)
STOP
```

(The following code is created and compiled separately.)

```
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
```

(The following code is created and compiled separately.)

```
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

```
DECLARE SUBROUTINE MSG
ARGUMENT = ""
LOOP UNTIL ARGUMENT = "END"
      GOSUB GETNUM
      IF VALID.NUM THEN GOSUB PRINTANSWER
REPEAT
STOP

GETNUM:
TEXT = "Enter an argument:"
MSG( TEXT, 'R', ARGUMENT, "")
VALID.NUM = NUM(ARGUMENT)
IF VALID.NUM THEN
      CALL FACTORIAL(ARGUMENT,ANSWER)
END ELSE
      TEXT = "Argument is not numeric!"
      CALL MSG( TEXT )
END
RETURN

PRINTANSWER:
TEXT = "Factorial of %1% is %2%"
MSG( TEXT, "", "", ARGUMENT:@vm:ANSWER )
RETURN
```

The following external subroutine must be compiled and cataloged separately.

```
SUBROUTINE FACTORIAL(X,A)
A = 1
FOR I = 1 TO X
      A = A * I
NEXT I
RETURN
```

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 ")".

Note 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)	FIELD	PWR
[ ] (bracket)	FIELDSTORE	QUOTE
ABS	FMT	REM
ALPHA	HASH	REPLACE
ATAN	ICONV	RND
BITAND OR XOR NOT	INDEX	SEQ
CALCULATE	INSERT	SERIAL
CHAR	INT	SIN
COL1( )	LEN	SPACE
COL2( )	LN	SQRT
COS	MATUNPARSE	STATUS( )
COUNT	MOD	STR
DIRLIST( )	NEG	SUM
DRIVE( )	NOT	TAN
EXP	NUM	TIME( )
EXTRACT	OCONV	TIMEDATE( )
DATE( )	INMAT( )	TRIM
DELETE	INP	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.

Note To execute a routine, Advanced Revelation looks first in SYSOBJ, then in VOC. If it finds the routine in SYSOBJ, any cataloged routine of the same name will be ignored.

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)
```

Note 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

The following sample program calls an external function, PERCENTAGE, to calculate the percentage that one value is of another.

```
DECLARE SUBROUTINE MSG
DECLARE FUNCTION PERCENTAGE

map         = ""
map<1>      = "RC"
base_amount = ""
delta       = ""
message     = "This program calculates the "
message<-1> = "percentage of change between"
message<-1> = "a base value and the amount of change."
message<-1> = ""
message<-1> = "Please enter a base value: "
MSG(message,map,base_amount,"")
message     = "Now, enter the amount that changed:"
MSG(message,map,delta,"")
change      = PERCENTAGE(base_amount,delta)
message     =  "The value ":base_amount:" changed by
message<-1> = ":change:" percent."
MSG(message)
STOP
```

The following code, defining the PERCENTAGE function, must be compiled separately, and must be cataloged before running the sample program above.

```
FUNCTION PERCENTAGE(base, delta)
rate = delta/base
rate = OCONV(100*rate, "MD0")
RETURN rate
```

Accessing Advanced Revelation tables

A group of 15 record (row)-oriented commands in R/BASIC allow you to access rows in Advanced Revelation or bonded tables. Using these commands, you can open files, read and write records or fields, lock and unlock tables and rows, and perform related table 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 table that can be attached using the Advanced Revelation ATTACH command.

Record-oriented commands can also access a special form of table called a *cursor*. For more information on cursors, see "Cursors" later in this chapter.

Opening a file

Before accessing a table, you must first open the file that contains 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"
SCRIPT = "WITH {ST} EQ 'WA' OR WITH {ST} EQ 'CA'"
REDUCE(SCRIPT, SORT_LIST, MODE, FILE_NAME, ⇨
      CURSORVAR, FLAG)
IF FLAG ELSE
   MSG("Failure in REDUCE")
   STOP
END
```

(continued)

```
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 secion on "Using Advanced Revelation with Networks" in the *User's Guide* 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"
      TEXT = "CUSTOMER file locked. R=retry,Q=quit"
      MSG(TEXT,"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 a 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 infor- mation, 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
   FSMSG()
   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
```

(continued)

```
        END ELSE
           * error condition for READ
           MSG("Error reading record %1%","","",@ID)
        END ; * read
        UNLOCK FILE_CUSTOMER, @ID ELSE
           /* if the record cannot be unlocked, call
              a system error handler */
           FSMSG()
        END ; * unlock
     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_*nnn* in the SYSINCLUDE file (where *nnn* 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:

```
$INSERT SYSINCLUDE, FSERRORS_100
DELETE FILE_CUSTOMER, @ID ELSE
      ERROR = @FILE.ERROR<1>
      IF ERROR = FS_REC_DNE$ 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
```

(continued)

```
      * 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
         TEXT = "Error %1% writing to file!"
         CALL MSG(TEXT, "", "", STATUS())
         OSCLOSE OUT_FILE
         STOP
      END
      OUT_OFFSET += LEN( BLOCK )

   UNTIL DONE REPEAT
   OSCLOSE OUT_FILE
   IF STATUS() THEN
      TEXT = "Error %1% closing file!"
      CALL MSG(TEXT,"","",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 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 RECORDCOPY, 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 includes 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 a new 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 "Transaction processing" in the *User's Guide*. 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.

Note 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.

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.

Note 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."

Debugger commands

Topic	Command	Summary
Program Control	[Esc] or END	Aborts program and returns to TCL
	X, EXECUTE	Executes a command as though at TCL *command*
	G	Go. Resumes program control
	OFF	Return to operating system
Source Code	$ or ?	Displays the current line and program name
	S	Displays the screen as it appeared before entering the debugger
	#	Shows your memory usage
	E*n*	Sets multistep line counter to *n*
	G*n*	Goes to line *n*
	L*n*	Lists *n* 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 *x* and prompts for any new value
	\x	Displays the content of *x* in hexadecimal code
Break Table	B@=*program*	Breaks if program name = *program*
	B$=*n*	Breaks if a line number equals *n*
	B*x* operator constant	Breaks when a variable, *x*, meets certain conditions
	K	Clears the break table
	K*n*	Clears the break table entry *n*
	D	Displays the break table
Trace Tables	T	Toggles the trace mode on or off
	T$	Traces the source of a program
	T@	Traces a program source
	T*x*	Traces a variable, where *x* is the variable

(continued)

Debugger commands (continued)

Topic	Command	Summary
Trace Tables *(continued)*	U	Clears the trace table
	U*n*	Clears the trace table entry
	D	Displays the trace tables
Program Stack	PS	Displays the program stack
	PSP	Display the program stack in a popup
Return Stack	RP	Display the return stack in a popup

$, ? 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 'OBJECT EXAMPLE' if (LINE) = 24
3 break in program 'OBJECT 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 OBJECT 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.

Note When using the networking version of Advanced Revelation, entering [Esc] or END will unlock all records that have been locked by the 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 SOURCE EXAMPLE
```

will allow the user to edit a program in the SOURCE file called EXAMPLE. When the edit is completed, control will return to the Debugger.

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 G*n* 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 K*n* where *n* is the break table number of the command.

Ln Display source code by typing L*n* where *n* is the number of lines to display, starting at the present line.

OFF Type OFF to log out of 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.

PSP Displays the program stack in a popup. Stored in the DEBUGGER_PS record in SYSPOPUPS.

RP Displays the return stack in a popup. Stored in the DEBUGGER_RS record in SYSPOPUPS.

For more information, see BREAK, DEBUG, EXECUTE, LOCK, and UNLOCK in the chapter "R/BASIC command reference," and the RUN and OFF commands in the chapter "TCL command reference".

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 CALL FSMSG() ; STOP
FOR @ID = 11 TO 20
   READ @RECORD FROM FILE_IV, @ID ELSE @RECORD = ""
PRINT @ID "L#3" : {INV_AMT} "R#10"
NEXT @ID
```

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 34K in length (64K in Advanced Revelation for OS/2).

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).

@LIST.ACTIVE The flag that indicates whether there is an active select list in cursor 0, and what type of list it is. Possible values for @LIST.ACTIVE are:

zero	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 1 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. A non-zero value in the following fields indicates which features are active. Field values are:

1	insert mode on
2	wrap line on
3	insert line on
4	indent line on
5	case insensitive search 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 of system variables 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.

@CRT MAXHIGH The maximum number of lines 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 SYSINCLUDE 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 false, the R/BASIC PRINT function will print the ASCII characters zero through 31 using their assigned graphics. If set to true, 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	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 variables 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 record that contains the character stream currently used in the CAPTURE 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 characters not yet executed that are currently being played back by the CAPTURE PLAY process.
@PLAYDELAY	A delay factor used to slow the CAPTURE PLAY process.
@PROG.CHAR	The last character typed except for keystrokes defined in these arrays: @INT.CONST, @PRIORITY.INT, EDIT.KEYS, and EXCEPT.KEYS.
@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 SYSINCLUDE 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 always active (example: [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:

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 SYSINCLUDE 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 SYSINCLUDE 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.

WINDOW_COMMON%

The window processor maintains a list of 135 COMMON variables known as the WINDOW_COMMON% variables. These variables are used to store information such as general window attributes and specific prompt attributes. By including the WINDOW_COMMON variables in your programs (subroutine), you can affect the way the window looks and 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).

Using WINDOW_COMMON%

You can only use the WINDOW_COMMON block if you are writing an external subroutine. You can then call this subroutine from anywhere in a window (most frequently, by using the "S" code and command, and specifying the name of your routine as the program to call.

To get access to the window common variables, you must insert them into your program. Always use this statement at the beginning of the program before attempting to read or change a value from one of these variables:

```
$INSERT SYSINCLUDE, WINDOW COMMON%
```

Some window common variables acce
WC_RESET% can be set to various int
Many of these values have equated con
RESET.PROMPT$ is equated to the va
program, use this statement at the top:

```
$INSERT SYSINCLUDE, WIN
```

NOTE TO THE UNWARY (IS THAT A WORD?)
THE EXAMPLES HAVE NUMEROUS "BUGS", SEE EXAMPLE FOR WC-BROWSE-NEXT% 2ND LINE, BROWSE-LIST SHOULD BE WC_BROWSE-LIST% NEW.BROWSE & RESET SHOULD HAVE WC_ & % ALSO NAMES ENDING WITH A $ SHOULD HAVE "." INSTEAD OF "_" ! (MANUAL WRITER MUST NOT DO MUCH PROGRAMMING

WINDOW_COMMON% examples

In the following pages, the most commo
are described. A short description of eac
program or program fragment illustrates

WC_AMV_ ACTION% The associated multivalue (A
the action taken on an AMV g
WC_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
5	An AMV other than the current AMV has been changed

The following code is an example (called as a Perpetual process):

```
IF WC_AMV_ACTION% = 2 AND MV = 1 THEN ⇨
   WC_AMV_ACTION% = 0
```

This will prevent the deletion of the first value in any AMV group in the window.

WC_BROWSE_NEXT% The WC_BROWSE_NEXT% variable indicates the next browse list to be processed. Keys are separated by value marks. For example:

```
WC_BROWSE_NEXT% = "KEY1":@VM:"KEY2"
BROWSE_LIST = WC_BROWSE_NEXT%
NEW.BROWSE = TRUE
RESET = RESET.RECORD$
```

WC_CURR_ AMV_ GROUP% The number of the current AMV group. If no AMV group is active this variable is null.

WC_DATA FILE% 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 WC_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.

WC_DISPLAY_ACTION% A code indicating how the system should display the current record. WC_DISPLAY_ACTION% can be used to force the redisplay of @RECORD or the recalculation of symbolic prompts.

WC_DISPLAY_ACTION% can take one of four values:

Value	Meaning
5	Redisplay all prompts
6	Redisplay the values of a list of prompts (WC_REDISPLAY_LIST)
7	Redisplay a list of prompts (WC_REDISPLAY_LIST). Prompts can be added or removed from the window.
9	Recalculate prompts (WC_RECALC%).

For example, as part of a pre-prompt process:

```
EQUATE REDISPLAY_PROMPT$ TO 5
IS="NEW DATA"
WC_DISPLAY_ACTION% = REDISPLAY_PROMPT$
```

WC_IS% The data in the current field. This can be used in conjunction with WC_VALID% to do data validation from post-prompt processes. For example, as a part of a post-prompt process:

```
IF WC_IS% NE IS.ORIG THEN
    WC_VALID% = FALSE
    CALL MSG("NO CHANGES ALLOWED")
END
```

WC_IS_DFLT% The default value of the current prompt. WC_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(WC_IS%) THEN WC_IS_DFLT% = 'NEW DATA'
```

WC_IS_ORIG% The original value of the data in the current field. That is, the data before any changes were made.

WC_MV% The current value position of the current AMV group.

WC_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
text = "Which value?|1 to %1%"
CALL MSG(text?,'R',REPLY,NUM.VAL)
IF NUM(REPLY) AND REPLY NE NUM.VAL THEN
    WC_MV_NEXT% = REPLY
    RESET = RESET.PROMPT$
END
```

WC_NEW_ BROWSE% Flag indicating whether a new browse list is active. Used in conjunction with WC_BROWSE_NEXT% to activate a new browse list.

WC_OREC% The value of the record as it was read off of disk. Compare @RECORD to WC_OREC% to see if any changes have been made. Useful in pre-read, pre-save, and pre-delete processes. For example:

```
IF @RECORD = WC_OREC% THEN...
```

WC_PKEY% The value of the previous key.

The following code fragment, called as a post-initialize process, establishes WC_PKEY% and WC_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 WC_PKEY% FROM SRC.DICT,'%SK%',1 ⇨
   ELSE STOP
WC_PKEY% = WC_PKEY% - 1
READ WC_PREC% FROM SRC.FILE,WC_PKEY% ELSE NULL
```

WC_PREC% The value of the record processed prior to the current record. WC_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:

```
EQU NULL$ TO ""
IF WC_OREC%=NULL$ AND WC_PREC% NE NULL$ ⇨
    THEN @RECORD<1>=WC_PREC%<1>
```

WC_REDISPLAY_LIST% A field mark delimited list of prompts to redisplay. If WC_DISPLAY_ACTION% is 6, the values of these prompts will be redisplayed. If WC_DISPLAY_ACTION is 7, only the prompts in WC_REDISPLAY_LIST% will be displayed in the window.

WC_RESET% A code used to tell the system what part of the process to reestablish. The code may take one of six values:

WC_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:

```
WC_RESET% = RESET_WINDOW$
```

WC_SI% A dynamic array containing the current prompt's characteristics. For example:

```
* put 100 in prompt register 1 for current prompt
WC_SI%<PREG1> = 100
```

WC_SRC_DICT% The file variable of the dictionary for the datafile used by the window.

WC_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 WC_SRC_FILE%, @ID ELSE
      CALL FSMSG()
   END
END
```

WC_VALID% If WC_VALID% is set to 0 (FALSE), then the current process is cancelled. It can be used when doing edit checks from a post-prompt process or as part of pre-read/save/delete/refresh processes.

The following is an example called as part of a pre-delete process:

```
IF @RECORD<1> NE "" THEN WC_VALID% = FALSE
```

Example called as part of a post-prompt edit check:

```
IF WC_IS% = 100 THEN WC_VALID% = FALSE
```

WC_W( )% A dimensioned array containing the prompt characteristics of all prompts on the window.

WC_W_CNT% The count of the number of prompts on the window. Used to dimension W.

WC_ WCHANGE% A flag indicating whether the data in the current field has been changed. True (1) indicates a change has been made.

WC_WI% The number of the current prompt. For example,

```
WC_W(WC_WI%)%<PREG1>
```

WC_W(WC_WI%)% is equivalent to:

```
WC_SI%<PREG1>
```

WC_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, WC_RESET% will have to be set. For example, the following illustrates WC_WI_NEXT% changed from a pre-prompt process:

```
IF WC_IS% THEN
   WC_WI_NEXT% = 3
   WC_RESET% = RESET_PROMPT$
END
```

This example illustrates WC_WI_NEXT% changed from a post-prompt process:

```
IF WC_IS% = WC_IS_ORIG% THEN WC_WI_NEXT% = 5
```

4. Building formulas

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 columns 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 column.

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 columns in the row (for example, an invoice total)
- formulas that string together (concatenate) values in columns, perhaps together with literal values (for example, a mailing label format)
- formulas that extract (translate) information out of related tables (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 columns 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 column (line) in the appropriate dictionary row. For more information about dictionaries, see the chapters "About databases in Advanced Revelation" and "Managing columns" in the *User's Guide*.

Accessing the Formula window

You may reach the Formula window from the Command, Dictionary, Maketable, and Batch Update windows or from Paint by way of the Dictionary.

To enter from the Command window, either press [Shift F3] or press [F6] (Softkeys) and select Option 3 from the Command window 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) or click <Options> at the *Formula* prompt.

To enter from the Maketable window, press [Shift-F6] or press [F6] (Softkeys) and select Option 6 from the softkey popup.

To enter from the Batch Update window, just enter the *Formula* prompt. This will automatically invoke the Formula window.

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) or click <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) or click <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) or click <Options> to access the More Expression 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.

Note 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 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 rows with shorter names than the original dictionary row. For example, the synonym CO with the formula

```
@ANS = {COMPANY}
```

would return the value of the column 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 table 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 column number 1 for customer 100 from the CUSTOMER table.

@ANS = column formula

This is used to perform a calculation on columns within a table. 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 row.

Glossary of R/BASIC formula trigger words

The trigger words in this glossary are organized in alphabetical order. However, 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 row, the current row key or the value returned by the formula. The variables @ID, @RECORD, and @ANS are available from the popup.

- @ID Key from the current row.
- @RECORD Data in the current row.
- @ANS The value returned when the formula is done.

For instance, you could check the value of a column in the current row by searching the contents of @RECORD.

For information on special variables, see "Special R/BASIC variables" in the "Programming with R/BASIC" chapter.

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.

column_comparisons

(see comparisons)

column_concatenation

(see concatenation)

column_formula

Use *column_formula* to perform an operation using two columns. The column_formula popup writes the trigger words "column operator column" to the Formula window. The column trigger word displays a list of column names. The operator trigger word specifies the type of operation used with the columns. For more information, see the column and operator trigger words.

column_logical

Use column_logical to combine two columns 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 columns meets the test condition. The word AND is used for a logical expression that is true if both columns meet the test condition. For more information, see AND and OR (Logical Expressions) in the "R/BASIC command reference" in R/BASIC.

column_math

Use column_math to perform a calculation on two columns. 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 row for a column.

@ANS = constant

This formula type is used to assign a constant value to a column.

@ANS = join

This formula type is used to display the value of a column in another table.

@ANS = column_formula

This formula type is used to perform an operation on two columns.

@ANS = value

This formula type is used to assign the value of an R/BASIC expression to a column. 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.

Note 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 column_logical)

math

(see column_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 column_logical, comparisons, concatenation, column_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 column_math, concatenation, comparisons, and column_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 row. 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.

R/BASIC Reference

5. R/BASIC command reference

Introduction

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 parameter.
- Examples showing you how 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".

Command reference

, !, 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 */.

```
* [characters]
! [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 of 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. Multiline comment block cannot be nested.

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

/*
All the elements of CUST (I) from 1 to the highest key
are written to the output file
*/
```

; (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 is counted. If *start* is a positive number, the substring will start at the character in that position counting from the start of the string. If *start* is a negative number, the substring will start at the character in that position counting from the end of the string. If *start* is 0 (zero) or 1 (one), the substring will start at the first character in the string.

length Normally, 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 *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 last 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.

Note COL1( ) and COL2( ) are only set when *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 *start* and *length* between brackets.

variable String that is the object of the assignment. That is, you will replace all or part of *variable* with the expression given in the syntax.

start Character position at which the string to replace begins. A value for *start* must be a number or an expression that yields a number. If *start* is a positive number, the substring will start at that spot in the character string. If *start* is a negative number, the substring will start at that number of characters from the end of the string. If *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 Number of characters from *start* to replace. *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 Yields a string that replaces the substring in the variable. If *expression* yields a string that is longer than the specified substring, then the total string length in the variable is increased. If *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.

{ } (Braces)

The { } (brace) operators allow you to return a value to your program from a dictionary record . The returned values are placed at the location of the calling functions, i.e., the brace operators.

```
{column_name}
```

column_name

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 field name is known and the position may be changed. When the field name is not known, the CALCULATE function is used. When the position 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
POSITION = 1
FIELD_NAME = 'COMPANY'
*
LOOP READNEXT @ID ELSE DONE = TRUE
UNTIL DONE DO
```

(continued)

```
      READ @RECORD FROM CST_FILE, @ID THEN
         PRINT 'Record    ':@RECORD
         PRINT '{ }       ':{COMPANY}
         PRINT 'CALCULATE ':CALCULATE(FIELD_NAME)
         PRINT ' < >    ':@RECORD<POSITION>
      END ELSE
         CLEARSELECT
         STOP 'No record'
      END
   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, POSITION, 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 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.

Note The 34K limit to program size (64K in Advanced Revelation for OS/2) includes inserted source code.

FILE_NAME Advanced Revelation file that contains the source code to be inserted. If *FILE_NAME* is not specified, the compiler will search the file from which the program is being compiled. *FILE_NAME* must be the literal filename and may not be surrounded by quotation marks.

RECORD_NAME 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 is inserted at this point in the program.

```
$INSERT BP,COMMON
```

The record COMMON in file BP would be inserted.

+++, ---, ***, ///, ::: (Special multivalue arithmetic operators)

The five operators +++, ---, ***, ///, and ::: allow addition, subtraction, multiplication, division, and concatenation of two dynamic arrays.

```
expression operator expression
```

expression Any R/BASIC expression that yields a dynamic array.

Note 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 = 1²1²1²1
B = 2²2²2²2²2²2
PRINT A+++B
```

The result will be 3²3²3²3²2²2. 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>
```

Using < >

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 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.

Note 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".

= (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 — 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 *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.

=, +=, -=, := (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.

Note 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.

=, +=, -=, := (Assigning matrix elements)

An assignment statement assigns a value (=, +=, -=, :=) to an individual matrix.

```
matrix(subscript) = expression
matrix(subscript) += expression
matrix(subscript) -= expression
matrix(subscript) := expression
```

matrix (subscript) 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.

@

Use the @ function with the R/BASIC PRINT statement to alter the screen display and printer output. The @ function can be used to:

- Clear portions of the screen.
- Position the cursor on the screen.
- Change the characteristics of text sent to the screen (color, bright, blink, etc.).
- Change the characteristics of text sent to the printer (fonts and font attributes).

```
PRINT [@(code):] [@(column, line):] [@(attribute):] ⇨
    [expression]
```

@(Code), *@(column, line)*, and *@(attribute)* can be used in any combination and listed in any order. If you use more than one @ function in the same PRINT statement, separate the functions with ":" (colon).

Note Use *@(code)* and *@(column, line)* only when sending PRINT statements to the screen. Do *not* send *@(code)* or *@(column, line)* to the printer.

@(code)

Use *@(code)* to clear portions of the screen or to move the cursor to the home position (column 1, line 1). Valid codes are:

Code	Function
@(-1)	Clears the entire screen and moves the cursor home.
@(-2)	Moves the cursor home.
@(-3)	Clears the screen from the current cursor position to the end.
@(-4)	Clears the screen from the current cursor position to the end of the line.

@(column, line) Use *@(column, line)* to move the cursor to a specific position on the screen.

Note *Column* and *line* must be within the column and line limits of the monitor, or the cursor position is unpredictable. This is especially true if you develop in high-density video display mode (43- or 50-line display) and then attempt to run in standard 25-line mode.

@(attribute) Use *@(attribute)* to change printer and display output characteristics, including fonts, font attributes, colors, highlight, blink, etc.

A list of attributes is available in the PRINT_CONSTANTS record of the SYSINCLUDE file. This record contains a series of negative integer values and equated values for the available character format attributes. Embed the equated value or the actual negative integer value for a specific attribute in your program.

For more information on using the PRINT statement to change printer and display output characteristics, see "PRINT" later in this chapter .

Note To use *@(attribute)*, you must make the PRINT_CONSTANTS record in the SYSINCLUDE table available to your program. Include the following line of code in your program:

```
$INSERT SYSINCLUDE, PRINT_CONSTANTS
```

expression Value being printed or displayed. See "PRINT" later in this chapter for more information on valid expressions.

Correct use of @ function

```
$INSERT SYSINCLUDE, PRINT_CONSTANTS

code  = -1
column = 40
line  = 10
attrib = DRED$
text  = "Washington, D.C."
PRINT @(code) : @(column, line) : @(attrib) : text
```

This program clears the screen and then displays Washington, D.C. on the screen beginning in column 40 of line 10. Assuming you are using a color monitor, the text displays in dark red. Because the program includes @(code) and @(column, line), output cannot be directed to the printer.

```
$INSERT SYSINCLUDE, PRINT_CONSTANTS

attrib    = BOLDON$
x         = 12
y         = 6
PRINTER ON
PRINT @(attrib) : (5*x + y) : " inches"
PRINTER OFF
```

*This program prints **66 inches** in bold type to the printer.*

ABORT

The ABORT statement forces a return to the Command window (or previous recursive TCL level).

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 the Command window or previous menu level.

```
IF X EQ "END" THEN ABORT
```

Control returns to the Command window or the previous menu level if variable X is equal to 'END'

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 simply unsigned.

```
ABS(expression)
```

expression Must evaluate to 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.

Note 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) wiithout 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.

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.

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 The 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.

Note 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 Must evaluate to 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
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.

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 long (32 bits in Advanced Revelation for OS/2). Hence 2 to the 64th (approximately 10 to the 20th) is the largest number that can be used by these functions (2 to the 32nd or 4GB in Advanced Revelation for OS/2).

These functions can be useful when using the OUT statement and the INP function.

BITAND Calculates the bit-wise logical AND of two integers.

BITOR Calculates the bit-wise logical OR of two integers.

BITXOR Calculates the bit-wise logical exclusive OR of two integers.

BITNOT Takes the decimal integer yielded by expression and calculates the bit-wise logical NOT of that number.

expression Must yield an integer.

Correct use of BIT functions

Expression1	Expression	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
```

(continued)

```
    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.

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 | numeric_expression
```

ON — A value of 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 — A value of 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.

numeric_expression — You can also use any numeric expression to specify the status of the Break key. If the value of the *numeric_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.

Note 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(expression)
```

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 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
```

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 subroutine | @variable | @matrix(index) [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 SETPROGRAM command in the "TCL command reference" chapter of the *Reference* manual.

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.

Note 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.

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

If *expression* evaluates true (non-zero), the statements are executed, and no subsequent CASE statement is 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 is less than 10, the program calls the subroutine EXPEDITE. If QTY is more than or equal to 10 but less than 100, the program calls the subroutine REORDER. Otherwise the program continues without calling either subroutine.

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.

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*.

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.

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.

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
```

(continued)

```
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.

CLEARFILE

The CLEARFILE statement deletes all records from a file, but retains the file in the system.

```
CLEARFILE filevar THEN|ELSE statements
```

filevar A file variable created by a previous OPEN statement.

THEN The statement(s) following THEN are executed if a file is cleared successfully.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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"
      TEXT = "Are you sure you wish to clear file %1%?"
      MSG(TEXT,"RC",RESP,FILE)
      IF RESP[1,1] EQ "Y" THEN
         CLEARFILE FILE_VAR THEN
            TEXT = "File ":FILE:" successfully cleared
            MSG(TEXT)
         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.

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 the TCL CLEARLIST command.

```
CLEARSELECT [cursorvar]
```

Caution! If you exit a READNEXT loop before exhausting the select list, you must clear the select list using CLEARSELECT. Otherwise, your results may be unpredictable.

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* is an integer value of 0 to 8, indicating the cursor to clear.

Note 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.

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.

Note 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.

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.

Note 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.

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.

Note For more information, see COLLENGTH. See also the TCL LIST command S option 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.

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 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

COMMON

The COMMON statement provides a common area for storing variables used by both the main program and external subroutines.

```
COMMON 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 the Command window 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.

Note 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.

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 ...]
```

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 the Command window 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 order of that allocation and, therefore, 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.

Note 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
```

The following code must be separately compiled and cataloged (run the TCL SETPROGRAM command), before the preceding program can run.

```
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 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 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.

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.*

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 String to be searched for the number of occurences 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 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.

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 Any valid combination of variables, functions, or literals. If more than one *expression* is specified, the *expressions* are delimited by commas.

Note 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.*

Note For more information, see CLEARDATA and PROMPT.

Correct use of DATA

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.*

```
DATA "CST","288","1","SMART-COMM","JOHNSEN"
PERFORM "UPDATE"
END
```

The VOC record UPDATE runs the program UPDATE that locates the old data in a multivalued field and replaces it with the new data.

```
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
```

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
```

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.

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.

Note 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.

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 SUB-ROUTINE

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. Do not use the name of an R/BASIC subroutine or intrinsic function when declaring a user-defined subroutine or function.

Note You cannot declare a subroutine or function to be the same name as an internal function (see that topic in the chapter "Programming with R/BASIC" for a list), or unpredictable results will occur.

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
```

The following code must be separately compiled and cataloged.

```
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.

DELETE

The DELETE statement deletes a record from a file.

```
DELETE filevar | CURSOR cursorvar ,key THEN|ELSE
    statements
```

filevar 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.

Note 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.

Note 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 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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
```

(continued)

```
      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.

DELETE( )

The DELETE function is used to delete the data contents of a specified field, value, or subvalue from a dynamic array. The corresponding delimiter is also deleted.

```
DELETE(array, field, value, subvalue)
```

array Dynamic array that is to be searched.

Note For information on dynamic arrays, see "Dynamic Arrays" in the chapter "Programming with R/BASIC".

field Position number of the field to be deleted. The *value* and *subvalue* values must be 0 (zero) to delete *field*.

value Position number of the value to be deleted. The *subvalue* value must be 0 (zero) to delete *value*, and *field* must be greater than 0 (zero).

subvalue Position number of the subvalue to be deleted. The values of *field* and *value* must be greater than 0 (zero).

Note 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)
```

(continued)

```
            WRITE NEW_RECORD ON FILE.CST, @ID ELSE
               TEXT = "Unable to write %1% on CST file!"
               MSG(TEXT,"","",@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.

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])...]
```

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.

Note 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 Integer identifying the number of rows in *matrix*. Numbers resulting from expressions will be truncated to their integer value.

column 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.

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 an expression yielding the full name (including path, if appropriate) of the operating system file.

Note 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"
TEXT<-1> = "Size = ":SIZE
TEXT<-1> = "Date = ":DATE
TEXT<-1> = "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
```

(continued)

```
* 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).

Note 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.

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.

ECHO

The ECHO statement either suppresses or enables display of input characters on the terminal screen.

```
ECHO ON | OFF | numeric_expression
```

ON This is the default setting. Any character input is 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.

numeric_expression You can also specify any numeric expression to set echo value. If the *numeric_expression* evaluates to true (non-zero), ECHO ON is set. If the value of the *numeric_expression* is false (zero) or null, ECHO OFF is set.

The PROMPT command may be used to suppress the input prompt.

Note 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.

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.

Note 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.

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.

You can use EQU for EQUATE.

```
EQUATE 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 — 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.

EXECUTE

The EXECUTE statement sends a command to the Command window for execution and then returns to the calling program.

```
EXECUTE command
```

command A valid TCL command, a cataloged program, or null. Null will display the Command window at the next recursive level. After EXECUTE makes a recursive call to the thje Command 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 the Command window:

- 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.

Note 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 FILES"
EXECUTE CMD
```

Produces a list of currently-attached files.

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 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:

```
EXP function value = (base e)**(expression)
```

Note 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

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 | FUNCTION name [(parameter ⇨
    [,parameter...])]
```

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.

parameter Parameters specify and identify any arguments required by the subroutine or function. Parameters are enclosed in parentheses, multiple arguments are delimited with commas.

Note 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
```

The following code must be separately compiled and cataloged.

```
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 Dynamic array that is to be accessed.

field Number of the field in *array* to extract. The *value* and *subvalue* values must be 0 (zero) to extract the *field*.

value Number of the value to be extracted. The *subvalue* value must be 0 (zero) to extract *value*. *Field* must be greater than 0 (zero).

subvalue Number of the subvalue to be extracted. The values of *field* and *value* must be greater than 0 (zero).

Compiled code will run more efficiently when field is an integer constant, and *value* and *subvalue* are equal to zero.

Note 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.

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 Source string for the FIELD operation.

delimiter The value of *delimiter* determines what in the string is considered a field. For example, in the string "REALNAME*REALVOL*REALACCNT" an asterisk determines that there are three fields in the string. 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 Which "field" is to be returned. If the occurrence *delimiter* evaluates to less than 1 (one), 1 (one) is assumed.

field Number of successive fields 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). For example, if STRING is:

"REALNAME*REALVOL*REALACCNT"

then FIELD(string, '*', 2, 2) will return

"REALVOL*REALACCNT".

Note 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.

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 String to be searched.

delimiter Any single ASCII character including field, value, or subvalue marks. This value determines what constitutes a "field" in the values of both *source_string* and *string*.

occurrence Processing starts at the substring designated by *occurrence*.

The value of *delimiter* determines the end of the first `field". If the number of substrings within *source_string* is less than the value of *occurrence*, *source_string* will be padded with the necessary number of null substrings before FIELDSTORE is executed.

field Number of substrings in *string* to apply to *source_string*.

As with *source_string*, the value of *delimiter* determines the end of a substring. 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 New string values to store in *source_string*.

Correct use of FIELDSTORE

The following routine changes a phone number, whereas the area code remains the same.

```
orig   = "206-111-2222" ;* old number
delim = "-"
occur    = 2 ;* start with 2nd field
field    = 2 ;* replace 2 fields
string   = "333-4444" ;* new number

new_number=FIELDSTORE(orig,delim,occur,field,string)
```

The following program prints the input string before and after the FIELDSTORE function.

```
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
```

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.

Note 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.

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 Any character string to format on output.

format An argument for *format* may be structured as follows:

```
justification{(fill_char)}mask
justification{(fill_char)}#{field_size}
conversion
```

justification All formats 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 Optional. Specifies the size of the field in which the value is to be justified. It must yield a positive integer. If the output data has a length larger than *field_size*, it will be truncated.

mask Template for 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 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. 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
```

(continued)

```
    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.

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 Must yield a legal R/LIST FOOTING string. Enclose the string in quotes.

Note See "Format keywords" in the "R/LIST keyword reference" chapter of the *Reference* manual.

options Included within the double quotes enclosing *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
{*column*}'	Insert the current value of *column* into the footing.

Time and date output formatting

You can formazt the output of heading and footing time and date information in the same way you can specify time and date display in a window, by including a output format specification. Use any format code that you would use at a window prompt (or in the R/BASIC OCONV statement).

The general syntax for formatting the time and date display is:

```
FOOTING "text 'option[,format]'"
```

where *format* is any format specification such as "D2/", "MT", etc. For details on what formats you can use, see the statement OCONV.

For example, the format "D4." produces output in the format "mm.dd.yyyy" So, to specify a footing that contains a date in that format, you would include that argument after a comma that immediately follows the T, D, or TD *option* value.

```
FOOTING "Output date: 'D,D4.'"
```

This would produce the footing

```
Output date: 09.30.1992
```

Note 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.

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 | UNTIL conditions [statements]]
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 *variable* to end the loop is an exception.

When the FOR/NEXT conditions have been satisfied, the value of *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 Gets assigned the current iteration value. Also called the loop variable or index variable.

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 An integer to assign as the beginning value of *variable*.

end 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 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.

Note *Amount* can be either a positive or a negative integer value.

WHILE conditions Limits or controls the iteration of the FOR ... NEXT loop. *conditions* can be any valid R/BASIC expression(s) If *conditions* evaluates to true (non-zero), the FOR ... NEXT loop is executed. If *conditions* is 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.

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 [parameter [, parameter] ...]
    statements
RETURN expression
```

function Any valid variable identifier.

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.

parameter Optional, and can contain a variable, matrix, constant, and expression. If used, all parameters must be defined within parentheses. If no parameter is specified, the parentheses are omitted. Multiple parameters are delimited with commas.

The parameters can be used to pass arguments 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 COMPOUND
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*COMPOUND(INT.RATE,PERIODS)
PRINT TOTAL.COST
END
```

Function Program

```
FUNCTION COMPOUND(INT.RATE, PERIODS)
RATE = (1 + (INT.RATE/100))**PERIODS
RETURN RATE
```

The program uses the declared function COMPOUND to calculate the compound interest and add it to SUBTOT. The result, TOTAL.COST, is printed.

GARBAGECOLLECT

The GARBAGECOLLECT statement forces a recovery of all unused or abandoned string space.

Note In Advanced Revelation for OS/2, GARBAGECOLLECT is unnecessary, and will have no effect.

```
GARBAGECOLLECT
```

String Space

R/BASIC constantly recovers string space at regulated intervals. GARBAGECOLLECT forces recovery of the available string space between these intervals.

Note GARBAGECOLLECT is ordinarily used only after a FLUSH statement.

GETBREAK, GETECHO, GETPRINTER, GETPROMPT

The GET functions return the current settings for BREAK, ECHO, PRINTER, and PROMPT system variables.

```
GETBREAK( )
GETECHO( )
GETPRINTER( )
GETPROMPT( )
```

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).

GET-ECHO( ) Returns the value set by ECHO. ECHO ON sets the value to 1 (one) and ECHO OFF sets the value to 0 (zero).

GET-PRINTER( ) Returns the value set by PRINTER. PRINTER ON sets the value to 1 (one) and PRINTER OFF sets the value to 0 (zero).

GET-PROMPT( ) 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 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.

GET.CURSOR

GET.CURSOR returns the current cursor position and the current cursor shape.

Note 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[1,1] )
ROW = SEQ( CURSOR.INFO[2,1] )
BOT = SEQ( CURSOR.INFO[3,1] )
TOP = SEQ( CURSOR.INFO[4,1] )

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
```

Using 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 Any valid numeric or alphanumeric label. An alphanumeric *label* must start with an alphabetic character.

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
```

(continued)

```
      AVG.CNT += 1
      IF RESPONSE NE TEST.LINE THEN
         PENALTY = 5
      END ELSE
         PENALTY = 0
      END
      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.

GO is a synonym for GOTO.

```
GOTO 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.

Note 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
```

HASH

The HASH function takes a record key, modulo, and offset and returns the group number as used in Advanced Revelation's 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.

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.
'Sn'	Spaces	Inserts `n' spaces into the heading at that point.
'T'	Time	Inserts the current time and date.
''	Two Single Quotes	Prints a single quote at that point.
{column}	Column name	Insert current value of *column*.

Time and date output formatting

You can format the output of heading and footing time and date information in the same way you can specify time and date display in a window, by including a output format specification. Use any format code that you would use at a window prompt (or in the R/BASIC OCONV statement).

The general syntax for formatting the time and date display is:

```
HEADING "text 'option[,format]'"
```

where *format* is any format specification such as "D2/", "MT", etc. For details on what formats you can use, see the statement OCONV in the *R/BASIC* manual.

For example, the format "D4." produces output in the format "mm.dd.yyyy" So, to specify a footing that contains a date in that format, you would include that argument after a comma that immediately follows the T, D, or TD *option* value.

```
HEADING "Output date: 'D,D4.'"
```

This would produce the heading

```
Output date: 09.30.1992
```

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 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
            CLEARSELECT
            STOP
         END
      END ELSE
         FLAG = "STOP"
      END
   REPEAT
END
```

ICONV

The ICONV function converts data into the internal system format used by Advanced Revelation. These conversions are known as input conversions.

```
ICONV(string, conversion)
```

Using ICONV

The point of specifying an ICONV conversion is to tell the system what kind of data is being input. If you specify a 'D' conversion, then it interprets that data as a date. If you specify an 'MS' conversion, then it interprets that data as a floating point value.

If you input a valid date value, but have set up the system to recognize a scientific notation value at that prompt, then if the system successfully converts the input value in accordance with the MS directive, the resulting internal value will have nothing to do with dates. Ordinarily, an input conversion pattern will not recognize data that is in an incorrect format. In fact, the ability for ICONV conversions to recognize only the proper input formats makes the ICONV statement an excellent means of input data validation, either at the prompt level or in the data dictionary.

Normally, if you specify that an input value be a European date, then a value that cannot be interpreted as a date at all will be rejected by the system. However, if you input that date in the wrong order (say, month/day/year), the system will try to interpret it as if it were in the correct order, resulting in an incorrect value. Any value that exceeds the specified domain, however, will be rejected.

For example, you specify a DE conversion. That presumes that the order of input data will be day/month/year. If the value 05/06/92 is input, the system interprets the input data as 5 June 1992, even if the user had intended it to mean 6 May 1992. But, such a value as 92/06/05 would be rejected by the system, because 92 is too great a value for the day-of-the-month domain it was expecting.

string Data to be converted to internal system format.

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 Identifies the type of data conversion that you want to perform on *string*.

If *conversion* is a literal value, you must put it in quotation marks. If *conversion* is a variable, do not enclose it in quotation marks.

Results

After executing an ICONV conversion, you can determine the success of the conversion by checking the value of STATUS( ). Possible values of STATUS( ) are:

STATUS( )	Meaning
0	Successful conversion.
1	The data in *string* cannot be converted using *conversion*.
2	*Conversion* is not a recognized conversion specification.

For further information on input conversions, see:

ICONV Boolean (B)
ICONV Date (D)
ICONV Datetime (DT)
ICONV Masked Decimal (MD)
ICONV Masked Scientific (MS)
ICONV MX, HEX, MO, MB
ICONV Time (MT)
ICONV Variable Binary (VB)

Correct use of ICONV function

```
X = ICONV(100.00,"MD2")
```

Converts 100.00 to 10000 and assigns the value to X.

```
D = ICONV("9-27-99","D")
```

Converts September 27, 1999 to internal date format, 11593, and assigns that value to D.

```
T = ICONV("1:00PM","MT")
```

Converts 1:00 PM to internal time format, 46800, and assigns that value to T.

```
DECLARE SUBROUTINE MSG
STRING      = ""
CONVERSION  = ""
EQUATE ESC$ TO CHAR(27)

LOOP
   TEXT = "ICONV"
   TEXT<-1> = "Enter the string to convert,"
   TEXT<-1> = " or [Esc] to quit"
   MSG(TEXT,"RE",STRING,"")
```

(continued)

```
WHILE STRING ESC$
   TEXT = "Enter a conversion pattern"
   MSG(TEXT, "R", CONVERSION, "")
   RESULT = ICONV(STRING,CONVERSION)
   IF STATUS() = 1 THEN
      TEXT = "ICONV error '%1%': Invalid data string."
      MSG(TEXT, "", "", STATUS())
   END ELSE
      IF STATUS() = 2 THEN
         TEXT = "ICONV error '%1%'"
         TEXT<-1> = "Invalid conversion pattern."
         MSG(TEXT,"","",STATUS())
      END ELSE
         TEXT = "ICONV"
         TEXT<-1> = "string: '%1%'"
         TEXT<-1> = "conversion: %2%'"
         TEXT<-1> = "result: '%3%'"
         PARMS = STRING:@VM:CONVERSION:@VM:RESULT
         MSG(TEXT,"","",PARMS)
      END
   END
REPEAT
```

This program prompts for a value for string and for conversion, and displays the results of successful conversions. If a conversion fails, the program displays the value of STATUS( ) and an error message.

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 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 Indicates a Boolean conversion. If no additional options are provided, the conversion examines the default Boolean conversion in the currently active language/national data set, and uses the first letter of the true string and the first letter of the false string found there. In the default language/national data set, "B" is interpreted the same as "B Yes,No".

BX Interprets Boolean input in accordance with the first value in the 9th field (Output Format) of the active LND record.

tf If *tf* is specified, 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.

If you do not specify values for either the *tf* or the *true,false* parameters, the default is the value in the active language set.

Note For more information, see "International applications" in the appendix to the *Reference* manual,

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 specifies how the system interprets input data, as it converts that data into the Advanced Revelation internal date format.

```
ICONV(string, "D [year] [char] [E] [F] [G] [H] [I] [J] ⇨
    [month_format] " )
```

Internal system date

Advanced Revelation saves all dates as the number of days before or after December 31, 1967. This date is saved as day 0 (zero).

Any date *after* December 31, 1967 is saved as a positive number, the number of days that have elapsed since that date. Any date *before* December 31, 1967 is saved as a negative number, the number of days backward from that date.

Interpreting input dates

An ICONV operator determines how input data is interpreted. For example, if the system can parse the input data into three elements (day, month, and year, in any order), then it will interpret which is which, according to the date conversion that is specified, or to the date conversion that is in effect (see the X option, below).

string Must contain a date in a format that is compatible with the conversion (see "E, F, G, H, I, J" below).

For a given conversion, you can enter values in a number of different ways. The day, month, and year can be separated by almost any non-numeric character. For example, for the conversion "DE", *string* can be any of the following values:

01 MAR 1992	01/mar/1992	01 Mar 92
01 March 1992	01 03 1992	1,3,92
1 3 92	01/03/92	1.3.1992
01 Mar 1992	1. MARCH 1992	March 1, 1992

You can enter the month as a number, an abbreviation, or fully spelled out. If you spell out or abbreviate the month, you must use the exact spelling (including punctuation, if any) in the active language set. For more information, see "International applications" in the appendix to the *Reference* manual.

If you do not specify a year, the year from the system clock is used. In addition, if you only specify two digits for the year:

30 to 99 are converted as 1930 to 1999.

00 to 29 are converted as 2000 to 2029.

Note If *string* is not in the correct format for the conversion, a null is returned.

D — Indicates a date conversion.

year and char — Have no effect on the ICONV function. These are OCONV options that are allowed in the ICONV function so you can use the same specifications for both input and output conversions. See "OCONV Date (D)" later in this chapter for more information.

E, F, G, H, I, J — Determine the order in which the system interprets the date (day, month, and year). See also the X option, below.

Format	Option
01 MAR 1992	DE
MAR 1992 01	DF
1992 01 MAR	DG
MAR 01 1992	DH
01 1992 MAR	DI
1992 MAR 01	DJ

Note If you enter the date in numbers, make sure you enter the date and month in the correct order for the option. For option "DE", 3/1/92 converts to 8769, while 1/3/92 converts to 8827. If you do not specify an option, and if you enter dates as numbers, option *H* is assumed.

month_format — Has no effect on the ICONV function. This is an OCONV option that is allowed in the ICONV function so you can use the same specifications for both input and output conversions. See "OCONV Date (D)" in this chapter for more information.

Note For more information, see "DATE( )" in this chapter, as well as "OCONV Date (D)" and "TIMEDATE( )".

X — Interpret input data in accordance with the Output Format, specified in the 9th field of the language set record format. This avoids restricting a format to a specific language set, and allows the conversion to match whatever language set is currently in effect.

Correct use of ICONV Date (D) function

```
DATE = ICONV("12/01/92","D")
CALL MSG(DATE)
```

Displays the message "9102".

```
CD = ICONV("01:01:93","D2:")
```

Assigns the value 9133 to the variable identifier CD.

The following table provides examples of the correct use of the ICONV Date function:

Example	Output
ICONV("1 Mar 92", "DE")	8827
ICONV("1/3/92", "DE")	8827
ICONV("3.1.92", "DE")	8769
ICONV("92*1*3", "DG")	8827
ICONV("March 1, 1992","DH")	8827

ICONV Datetime (DT)

The ICONV datetime function converts date and time string into an internal decimal number.

```
ICONV(expression,"DT | DTX [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 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 the date portion of *expression*, you can enter the month as a number, an abbreviation, or fully spelled out. If you abbreviate or spell out the month, you must use the exact spelling (including punctuation, if any) in the active language set.

If the time portion of *expression* is in 12-hour format, you must use the abbreviations found in the active language set (for example, AM and PM in English).

Note For more information about language sets, see the "International applications" appendix in the *Reference* manual.

For more information on date and time conversions, see the appropriate ICONV function descriptions.

DT 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.

DTX Datetime conversion, but the specific format is taken from the 9th field of the language set record format. See the "International applications" appendix to the *Reference* manual,

dateconv Any valid date conversion, as documented under the heading ICONV Date (D).

Note 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 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 Any valid time conversion as documented under the topic ICONV Time (MT).

Note 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( TIMEDATE(), "DT" )
```

TEST is set to the internal (fractional) date and time at which the function is executed.

ICONV MX, HEX, MO, MB

The ICONV function with MX, HEX, MO, or MB conversion specifications converts values from hexadecimal, octal, or binary formats to decimal numbers. The value being converted may be up to 64 bits long (32 bits in Advanced Revelation for OS/2).

```
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.

Note 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

ICONV Masked Decimal (MD)

The ICONV function with Masked Decimal (MD) conversion enables you to convert decimal numbers into integers, Advanced Revelation's internal format for saving numbers.

MC is equivalent to MD, simply substituting a comma as the decimal point.

```
ICONV(string, "MDn [scale] [,] [$ | m | [mon]] [- | < | ⇨
    C | D]  [P] [Z] [T] [S] [xc]")
```

Caution! Do not allow spaces between arguments. See "Correct Use of ICONV Masked Decimal (MD) Function" below.

string Must be either a number or a variable that contains a number.

MD, MC MD or MC indicates a masked decimal conversion.

MC (Masked Comma) uses a "." (period) to separate thousands and a "," (comma) to separate the fractional portion of a decimal number.

Note If you use the MC option, and if the value of *string* is a literal value (rather than a variable name), you must enclose *string* in quotation marks.

n A single digit, 0-F, indicates how many digits after the decimal to retain. Excess digits are rounded off. The decimal point does not appear in the data. A value for *n* is required.

scale A single digit, 0-9, indicates a scaling factor. The default is the value of *n*.

If you specify a value for *scale*, *string* is multiplied by 10 to a factor of *scale*. When you use the OCONV function to convert *string* for output, *string* is divided by 10 to a factor of *scale*.

Note For more information on scaling factor, see "OCONV Masked Decimal (MD)" later in this chapter.

Remaining options

The remaining options have no effect on the ICONV function. They are OCONV options that are allowed in the ICONV function so you can use the same specifications for both input and output conversions. See "OCONV Masked Decimal (MD)" later in this chapter for more information.

Note If *string* includes a multi-character currency symbol (for example, "Fr" for francs), you must specify that currency symbol in the ICONV conversion. See "$, m, or [mon]" under "OCONV Masked Decimal (MD)" later in this chapter, for more information.

Correct use of ICONV Masked Decimal (MD) function

```
CELSIUS = 31.4
TEMP = ICONV(CELSIUS,"MC2")
CALL MSG(TEMP)
```

Displays the message "31400".

The following table provides examples of the correct use of the ICONV Masked Decimal function:

Example	Output
ICONV(64.24,"MD2")	6424
ICONV(-5,"MD1")	-50
ICONV(64.24,"MD3")	64240
ICONV(143.5,"MD12")	14350
ICONV(143.5,"MD14")	1435000
ICONV(143.5,"MD0")	144
ICONV("1.234,5","MC2.")	123450
ICONV("$45.50","MD2")	4550
ICONV("45,25Fr","MC2[Fr]S")	4525

ICONV Masked Scientific (MS)

The ICONV function with Masked Scientific (MS) conversion enables you to convert scientific notation into Advanced Revelation's internal storage format.

```
ICONV(string, "MS [e mm ] [,] " )
```

string — Either a number in valid scientific notation or a variable that contains a number in valid scientific notation.

MS — Indicates a scientific notation conversion.

MSX — Interprets input data as scientific notation in accordance with the 6th value of the 9th field (Output Format) of the active LND record.

e and mm — Have no effect on the ICONV function. These are OCONV options that are allowed in the ICONV function so you can use the same specifications for both input and output conversions. See "OCONV Masked Scientific (MS)" later in this chapter for more information.

, (comma) — If you use the "," (comma) option, you can use a comma instead of a period in *string* to indicate a decimal number.

Note If you use the "," option, and if the value of *string* is a literal value (rather than a variable name), you must enclose *string* in quotation marks.

For more information, see "OCONV Masked Scientific (MS)" later in this chapter.

Correct use of ICONV Masked Scientific (MS) function

The following table provides examples of the correct use of the ICONV Masked Scientific function:

Example	Output
ICONV(1.23,"MS")	1.23
ICONV(1.2345E+4,"MS")	12345
ICONV(1.23456789E+2,"MS")	123.456789
ICONV(1.234E+5,"MS")	123400
ICONV(1234.1234E-2,"MS")	12.341234
ICONV("1234,1234E-2","MS,")	12.341234

Values for MSX conversions depend upon the active language set.

ICONV Time (MT)

The ICONV function with Time (MT) conversion enables you to convert times into Advanced Revelation's internal storage format.

```
ICONV(string, "MT [H] [S] [char]")
```

Internal system time

Internal system time is calculated as the number of seconds past midnight. A 24-hour day has 86,400 seconds.

string Either a value in a valid time format or a variable that contains a value in a valid time format. You can use any character between ASCII characters 33 and 248 inclusive to separate hours, minutes, and seconds.

Note If the value of *string* is in 12-hour format, you must use the abbreviations found in the active language set (for example, AM and PM in English). For more information, see "International applications" in the appendix to the *Reference* manual,

MT Indicates a time conversion.

MTX Interprets input data as a time conversion in accordance with the 3rd value of the 9th field (Output Format) of the active LND record (language set).

H, S, and char

The *H*, *S*, and *char* options have no effect on the ICONV function. These are OCONV options that are allowed in the ICONV function so you can use the same specifications for both input and output conversions. See "OCONV Time (MT)" later in this chapter for more information.

Correct use of ICONV Time (MT) function

```
TIME = "08:28PM"
CONVERSION  = "MTH"
INTERNAL_TIME = ICONV(TIME,CONVERSION)
```

The value 73680 is saved in INTERNAL_TIME.

The following table provides examples of the correct use of the ICONV Time function:

Example	Output
ICONV("12:01","MT")	43260
ICONV("12:01PM","MTH")	43260
ICONV("12:01:59PM","MT")	43319
ICONV(2,"MT")	7200
ICONV("2:01AM","MT")	7260
ICONV("2/01/59","MT,")	7319

ICONV Variable Binary (VB)

The Variable Binary (VB) conversion for ICONV and OCONV enables storage of binary data that might include Advanced Revelation system delimiters.

```
ICONV(string, "VB")
```

Some possible byte values (ASCII 0 and ASCII 249-255) cannot normally exist as data within an Advanced Revelation environment, without risking confusion with their functions as system delimiters. If such values as ASCII 253 and 254 were allowed to exist as data in an Advanced Revelation environment, they would be interpreted by Advanced Revelation tools in their roles as data delimiters. This conversion mode stores such values in a safe form that cannot be misinterpreted by the system. Use OCONV to restore these values to their single-byte values.

The following values are affected by the VB conversion:

Char	Converted to
CHAR(0)	CHAR(0) CHAR(0)
CHAR(249)	CHAR(0) CHAR(1)
CHAR(250)	CHAR(0) CHAR(2)
CHAR(251)	CHAR(0) CHAR(3)
CHAR(252)	CHAR(0) CHAR(4)
CHAR(253)	CHAR(0) CHAR(5)
CHAR(254)	CHAR(0) CHAR(6)
CHAR(255)	CHAR(0) CHAR(7)

string Can contain any possible values, and is intended to contain binary data that has been generated outside the Advanced Revelation environment. By using the ICONV VB conversion, you can store this data within Advanced Revelation, process it with Advanced Revelation tools, and then reconvert it to its original form (with changes) by using the OCONV VB conversion.

Correct use of ICONV Variable Binary function

The following values are converted from single-byte values to their corresponding internal format:

```
variable = \0D0A00FEFFFCFD\
new_value = ICONV(variable, "VB")
```

where the contents of *new_value* is now:

```
OD 0A 00 00 OO 06 00 07 00 04 00 05
```

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.

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.

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
```

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.

Note 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

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

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.

INITDIR

The INITDIR statement uses a operating system pathname to indicate a directory and files to be accessed by the DIRLIST function.

```
INITDIR expression
```

Using 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 "?".

Note 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
```

(continued)

```
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.

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).

Note 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
```

(continued)

```
   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.

INP

The INP function inputs a value from a port. An understanding of the hardware is necessary to use this function effectively.

Note INP is not currently implemented for OS/2

```
INP(port)
```

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.

Note 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.

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] [:] [_]
```

Using 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
```

Note 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 — 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
```

(continued)

```
   LOOP
      INPUT INP,-1
   UNTIL INP REPEAT
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.

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 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)
```

Note 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
   TEXT = "Enter field position (type END to quit)
   PRINT TEXT : ; 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 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)
```

Note 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.

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 *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).

LINEMARK

The LINEMARK statement allows the debugger to be called at specific lines in R/BASIC programs compiled with the L option.

```
LINEMARK
```

Using 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)
```

Using LN

LN calculates the natural logarithm to the (base e) of the value of *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.

LOCATE

The LOCATE statement searches a string to locate the position (index) of a substring that contains specified data. Substrings are separated by a delimiter character.

```
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.

Note 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 "")
```

(continued)

```
FOR I = 1 TO CNT
   PRINT CUST.NR<1>, TOT<1>
NEXT I
END
```

The program accumulates totals using dynamic arrays.

Note See "International applications" in the appendix to the *Reference* manual, for more information about language sets.

LOCATE BY

The LOCATE statement may be used with an optional BY clause for use when the data you are locating does not yet appear in the string being searched. This command is intended to be used with sorted strings. LOCATE BY returns a location where the data would be if it were inserted into the sorted string.

```
LOCATE substring IN string BY seq [USING delim] SETTING ⇨
    var THEN|ELSE statements
```

The BY clause in the R/BASIC LOCATE BY statement locates the sorted position (index) of a value. If your active language set is not DEFAULT (which uses the ASCII sort order), sorts resulting from a LOCATE BY statement are based on the character sort order specified in the active language set. If, for example, a French language set is assigned and you use the LOCATE BY statement in a program, the sort order is based on the French language set rather than on DEFAULT.

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)

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).

BY Clause

The BY clause is used if the search data does not appear in the string (in other words, if the ELSE branch is taken). LOCATE then returns a value in the *var* parameter that indicates where *substring* would appear in *string* if *string* were sorted in *seq* order.

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: ; * both 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
```

Using 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. For general information on locking in Advanced Revelation, see the chapter "Locking" in the *User's Guide*.

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 RESETSYSTEM command or the END command at the debugger prompt.

Note 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 — File variable for 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 — 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 *Networks*.

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.

Note 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 "Accessing Advanced Revelation tables" 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
      DONE = TIME() + 2
      LOOP UNTIL TIME() = DONE
      REPEAT
   END
UNTIL LOCKED REPEAT
```

This program fragment loops until the target record is locked. Note the use of an timed 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, FSMSG
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
            FSMSG()
         END
         UNLOCK CUST_FILE, KEY ELSE
            FSMSG()
         END
      END ELSE
```

(continued)

```
            IF STATUS() THEN
               MSG("Locked by you")
            END ELSE
               TEXT = "Locked by another station"
               TEXT<-1> - "[Esc] to quit, "
               TEXT<-1> = "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
```

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 | UNTIL test [DO statements]
    [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.

Note 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 = expression | MAT matrix2
```

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 *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.

Note 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
   MATRIX2 (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.

MATCHES

The MATCHES operator compares a string value to a predefined pattern and evaluates to 1 (one), if true, and 0 (zero), if false.

```
string MATCH | MATCHES pattern
```

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
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

Note 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 variable INTO matrix [USING delimiter]
```

variable Variable designates the dynamic array that contains the fields to be parsed into the matrix.

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 i*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"
```

(continued)

```
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.

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.

Note 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.

Note For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

filevar File variable from previous 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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, FSMSG
OPEN "CUSTOMER" TO CUST_FILE ELSE
   MSG("201","","","CUSTOMER") ;* file not attached
   STOP
END
KEY = "C932"
DIM CUST(30)
MATREAD CUST FROM CUST_FILE, KEY ELSE
   FSMSG()
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 Any 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.

MATWRITE

The MATWRITE statement writes the values of the elements of a designated matrix into a record.

```
MATWRITE matrix ON filevar | CURSOR cursorvar ,key ⇨
    THEN|ELSE statements
```

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.

Note For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

filevar File variable from previous 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*.

Note 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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.

MOD

The MOD function returns the modulo remainder of two expressions.

```
MOD( expression, expression )
```

Using MOD

The modulo (remainder) is calculated by the following formula:

```
MOD(X,Y) = X-(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".

NEG

The NEG function returns the inverse of a specified number.

```
NEG(expression)
```

expression Must evaluate to be numeric, and must not contain any special characters other than the "-" (minus sign), E scientific notation, and "." (decimal point).

If *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

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 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).

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
```

Using 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 CHRG FROM INV ELSE NULL
```

If no CHRG 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.

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 *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 "
   END ELSE
      PRINT "DATA ":STRING: " IS ALPHA "
   END
REPEAT
END
```

OCONV

The OCONV function converts data from the internal system format used by Advanced Revelation to a corresponding output format. These conversions are known as output conversions.

```
OCONV(string, conversion)
```

string Data to be converted to output format.

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 Type of data conversion that you want to perform on *string*.

If *conversion* is a literal value, you must put it in quotation marks. If *conversion* is a variable, do not enclose it in quotation marks.

Format specifications used in FMT may be used as conversions. See "FMT" for more information.

Results

After executing an OCONV conversion, you can determine the success of the conversion by checking the value of STATUS( ). Possible values of STATUS( ) are:

STATUS( )	Meaning
0	Successful conversion.
1	The data in *string* cannot be converted using *conversion*.
2	*Conversion* is not a recognized conversion specification.

For further information on output conversions, see:

OCONV Boolean (B)
OCONV Date (D)
OCONV Datetime (DT)
OCONV Masked Decimal (MD)
OCONV Masked Scientific (MS)
OCONV MX, HEX, MO, MB, MOX (monetary)
OCONV Time (MT)
OCONV Variable Binary (VB)

Correct use of OCONV function

```
TODAY = OCONV(DATE(), "D2-")
```

Converts today's internal date to output in the format MM-DD-YY and assigns the output to the variable TODAY.

```
PRINT OCONV(100000,"MD2$")
```

Prints $1000.00.

Note See "ICONV" earlier in this chapter for a sample program that prompts for a value for *string* and *conversion*, displays the results of successful conversions, and displays error messages for failed conversions.

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 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 Specifies a Boolean conversion. If no additional options are provided, the conversion examines the default Boolean conversion in the currently active language/ national data set, and uses the first letter of the true string and the first letter of the false string found there. In the default language/national data set, "B" is interpreted the same as "B Yes, No".

BX Specifies a Boolean conversion that uses the conversion format stored in the first value of the Output Format field (9) in the active LND record (language set).

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.

If you do not specify values for either the *tf* or the *true,false* parameters, the default is the value in the active language set.

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

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 converts a date in Advanced Revelation's internal date format into a date in most common formats.

```
OCONV(expression, "D [year] [char] [E] [F] [G] [H] [I]⇨
    [J][month_format] " )
```

Internal system date

Advanced Revelation saves all dates as the number of days before or after December 31, 1967. This date is saved as day 0 (zero).

Any date *after* December 31, 1967 is saved as a positive number, the number of days that have elapsed since that date. Any date *before* December 31, 1967 is saved as a negative number, the number of days backward from that date.

expression Must evaluate to either a positive or negative integer.

D D indicates a date conversion.

year The *year* option indicates the number of digits in the year. The range of valid values is 0-9. 0 (zero) indicates that no year is output. The default value is 4. If you specify a value greater than 4, the year is prefixed with 0s.

char The *char* option identifies the character that separates the month, day, and year. The default is a space. If you use the *char* option, the month is output as a two-digit number, unless you use the Medium or Long month formats. If you use the M or L month formats, the value of *char* will only be used to separate the day from the month or year.

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

E, F, G, H, I, J, X The *E, F, G, H, I, J,* and X options determine the order in which the day, month, and year are output. The default value is *E* if no separator character (*char*) is specified and *H* if a separator character *is* specified.

The X option derives its format from the active language set, so its output depends upon the currently loaded set. Conversions using X will automatically adjust to the active language set..

Option	Format
DE	01 MAR 1992
DF	MAR 1992 01
DG	1992 01 MAR
DH	MAR 01 1992
DI	01 1992 MAR
DJ	1992 MAR 01

month_format Specify how completely the date is displayed. Short month format ("S") uses only numbers for the month. Medium month format ("M") uses an abbreviated month name, and long month format ("L") displays the complete month name. The month name is taken from the active language set.

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

If you specify a separator character *char*, the default for *month_format* is "S". If you do *not* specify a separator character, the default for *month_format* is "M".

Month_format also determines where the separator character *char* is placed in the date. For *month_format* "S", the separator character appears between all parts of the date. For *month_format* "M" or "L", the character appears immediately after the day, unless the day is the last part of the date (conversions F and J).

The following table demonstrates the effects of *month_format*. In each case, *expression* is -699:

month_format	Example conversion	Output
none	"D4/H"	01/31/1966
S	"D4/HS"	01/31/1966
M	"D4,HM"	JAN 31, 1966
L	"D4,HL"	January 31, 1966

Note For more information, see "DATE( )","ICONV Date (D)", and "TIMEDATE( )" in this chapter.

Correct use of OCONV Date (D) function

```
INTERNAL_DATE = -699
CONVERSION  = "D2/"
DATE = OCONV(INTERNAL_DATE, CONVERSION)
```

Assigns the value 01/31/66 to DATE.

The following table provides examples of the correct use of the OCONV Date function:

Example	Output
OCONV(-699, "D")	31 JAN 1966
OCONV(-699, "D2")	31 JAN 66
OCONV(-699, "D4")	31 JAN 1966
OCONV(-699, "D/")	01/31/1966
OCONV(-699, "D2/")	01/31/66
OCONV(-699, "D*")	01*31*1966
OCONV(-699, "D/E")	31/01/1966
OCONV(-699, "D2E")	31 JAN 66
OCONV(-699, "D2/E")	31/01/66
OCONV(-699, "DS")	31 01 1966
OCONV(-699, "D/S")	01/31/1966
OCONV(-699, "DM")	31 JAN 1966
OCONV(-699, "D,M")	JAN 31, 1966
OCONV(-699, "DL")	31 January 1966
OCONV(-699, "D,L")	January 31, 1966
OCONV(-699, "DJL")	1966 January 31
OCONV(-699, "D4.EM")	31. JAN 1966
OCONV(-699, "D2.EL")	31. January 66

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.

Note 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 Specifies the datetime conversion. If no options are specified, the conversion "DT^1 " is assumed.

DTX For its output format, takes the 4th value of the Output Format field (9) in the active LND record. Is therefore valid for whatever language set is currently loaded.

dateconv Any valid date conversion as documented under the heading OCONV Date (D).

Note Do not include the "D" in the *dateconv* expression. Include only the optional parameters.

If the output conversion that you have chosen for *dateconv* displays either an abbreviated or a full month name, the month name is taken from the active language set. The characters used to indicate AM and PM are also identified by the active language set.

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

^ (caret) Use the ^ symbol (caret) to delimit the date conversion specification from the time conversion specification.

n Number of characters that will separate the date from the time. The default is one.

c Use 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 Any valid time conversion as documented under the topic OCONV Time (MT).

Note 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

OCONV MX, HEX, MO, MB, MOX

The OCONV function with MX, HEX, MO, or MB conversion specifications converts values from decimal formats to hexadecimal, octal, and binary formats. The result may be up to 64 bits long (32 bits in Advanced Revelation for OS/2).

```
OCONV(expression, "MX")
OCONV(expression, "HEX")
OCONV(expression, "MO")
OCONV(expression, "MB")
OCONV(expression, "MOX")
```

expression Must yield a decimal value. Scientific notation is accepted.

MX *expression* must be a decimal number (base 10) and is converted to the equivalent hexadecimal character set (base 16).

HEX *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 *expression* must be a decimal number and is converted to an ASCII representation of an octal (base 8) number.

MB *expression* must be a decimal number and is converted to an ASCII representation of a binary (base 2) number.

MOX *expression* will be converted to the monetary format specified in the 5th value of the 9th field of the active LND record (language set).

If expression yields an incorrect value a 0 (zero) will be returned.

Note 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, MOX 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.

```
A = internal_money_value
B = "MOX"
C = OCONV(A,B)
```

A value stored in the system internal format is converted to whatever monetary format is valid for the currently loaded language set.

OCONV Masked Decimal (MD)

The OCONV function with Masked Decimal (MD) conversion enables you to convert a decimal value from Advanced Revelation's internal format into the corresponding output format.

```
OCONV(string, "MDn [scale] [,] [$ or m or [mon]] ⇨
    [- or < or C or D] [P] [Z] [T] [S] [L] [xc]")
```

Caution! Do not allow spaces between arguments. See "Correct use of OCONV Masked Decimal (MD) Function" below.

string — either a number or a variable that contains a number. The value can be a decimal number.

MD, MC — Masked decimal conversion.

MC (Masked Comma) uses a "." (period) to separate thousands and a "," (comma) to separate the fractional portion of a decimal number.

n — The *n* option indicates the number of digits after the decimal point. Use the values A - F to specify 10 to 15 digits. The maximum value of *n* is F (15 digits). A 0 (zero) indicates no decimal point.

Note You must specify a value for *n*.

scale — A single digit, 0-9, indicates the scaling factor. The default value of *scale* is the value of *n*.

If you specify a value for *scale*, *string* is divided by 10 to a factor of *scale*. If you use the *P* option, *scale* is not used.

Use the scaling factor to adjust the results of multiplication or division on numbers that are already in internal format. For example, if you multiply 10.00 (internal format 1000) by 0.15 (internal format 15), the product should be 1.50. However, the internal format of the product is 15000. To assign the correct number of decimal places to the result, you must indicate a value for *n* of 2 (to display two decimal places). In addition, you must specify a scaling factor of 4 to cause the value in internal format to be divided by 10 to the 4th power.

Note Remember that when you multiply two decimal numbers, the number of decimal places in the product is the sum of the number of decimal places in the two numbers. This number is the value of the scaling factor. For example, if you multiply 100.00 by 0.15, the result is 1.5000 — a number with four decimal places.

, (comma) — The "," (comma) option inserts commas to separate thousands. If you are using the MC option, substitute a "." (period) for the comma.

$, m, or [mon] — There are three ways to indicate monetary units:

- Specify "$" (dollar sign) to prefix a "$" to the value.
- Specify any other single, non-reserved character to prefix that character to the value.
- Specify a series of characters, enclosed in square brackets, to prefix those characters to the value.

Note To suffix the monetary unit instead of prefixing it, use the *S* option.

–, <, C, or D There are four ways to indicate a negative value:

- Specify a "–" (minus sign) to suffix a "–" to negative values. A positive value is suffixed with a single space.
- Specify an "<" (angle bracket) to surround negative values with "<>". A positive value is suffixed with a single space.
- Specify *C* to suffix negative values with one or more characters that indicate CREDIT. The characters actually displayed are identified in the active language set. A positive value is suffixed with as many spaces as the number of characters in the abbreviation for "credit" in the active language set, typically two ("CR").
- Specify *D* to suffix negative values with one or more characters that indicate DEBIT. The characters actually displayed are identified in the active language set. A positive value is suffixed with as many spaces as the number of characters in the abbreviation for "debit" in the active language set, typically two ("DB").

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

P If you specify *P* and *scale*, and if *string* contains a decimal point, output is formatted to the number of decimal places indicated by *n*, but the location of the decimal place is retained.

Z The *Z* option outputs a value of 0 as null.

T The *T* option truncates data rather than rounding when applying decimal (*n*) and scaling (*scale*) factors.

S The *S* option causes the monetary unit, if any, to be suffixed rather than prefixed to the value.

L If the absolute value of the number to be output is less than 1, the *L* option prefixes a 0 before the decimal point. If the number is 0, output is 0.0. The *Z* option overrides the *L* option.

xc Use the *xc* option to specify a fixed length for output and a fill character in case output is shorter than the fixed length. *X* is the length of the output and may be one or more digits. Use *c* to specify the character used to pad the output to the length specified by *x*.

Caution! If the length of the formatted output exceeds *x*, output is truncated

Note If you use the xc option, you must separate the values for n and x with at least one of the following:

- A value for *scale*.
- Any non-numeric option.

Correct use of OCONV Masked Decimal (MD) function

The following table provides examples of the correct use of the OCONV Masked Decimal function:

Example	Output
OCONV(1234,"MD0")	1234
OCONV(1234,"MD2")	12.34
OCONV(12,"MD2")	.12
OCONV(150000,"MD24")	15.00
OCONV(12,"MD2L")	0.12
OCONV(1234,"MD2$")	$12.34
OCONV(-1234,"MD2$")	12.34
OCONV(-1234,"MD2$C")	$12.34CR
OCONV(-1234,"MD2$D")	$12.34DB
OCONV(-1234,"MD1-")	123.4-
OCONV(1234,"MD2,10*")	*****12.34
OCONV(-1234,"MD2$,10*-")	$***12.34-
OCONV(500,"MD0£")	£500
OCONV(50,"MD0[DM]")	DM50
OCONV(50,"MD0[DM]")	DM 50
OCONV(252525,"MC2.[Bfr]S")	2.525,25 Bfr
OCONV(-12345,"MC1.")	-1.234,5

OCONV Masked Scientific (MS)

The OCONV function with Masked Scientific (MS) conversion enables you to convert data in Advanced Revelation's internal storage format into scientific notation.

```
OCONV(string, "MS [e mm ] [,] " )
```

string Must be either a number or a variable that contains a number. The value can be a decimal number. If *string* is not a number, 0 (zero) is returned.

MS Specifies a scientific notation conversion.

MSX Formats output into the scientific notation format specified in the 6th value of the Output Format field (9) of the active LND record (language set).

e The number of digits in the power of 10 to which the number is raised. The default is 2. The maximum value is 9.

mm If you use the *e* option, you must use the *mm* option to indicate the number of digits in the mantissa, or the number of places following the decimal point. The default for *mm* is 6, and the maximum is 14. *Mm* may be one or two decimal digits.

, (comma) The "," (comma) option substitutes a "," for the decimal point.

Note For more information, see "ICONV Masked Scientific (MS)" earlier in this chapter.

Correct use of OCONV Masked Scientific (MS) Function

The following table provides examples of the correct use of the OCONV Masked Scientific function:

Example	Output
OCONV(1.23,"MS")	1.230000E+00
OCONV(-1.23,"MS22")	1.23E+02
OCONV(123,"MS15")	1.23000E+2
OCONV(123456,"MS23")	1.235E+05
OCONV(123456,"MS23,")	1,235E+05

OCONV Time (MT)

The OCONV function with Time (MT) conversion enables you to convert data in Advanced Revelation's internal storage format into times.

```
OCONV(expression, "MT [H] [S] [char]")
```

expression *Expression* must be either an integer or a variable that contains an integer.

Note Internal system time is saved as the number of seconds past midnight. A 24-hour day has 86,400 seconds. Any value over that number is divided by 86,400, and the remainder is the value of *expression*.

MT Specifies a time conversion.

MTX Formats an internal time value into the output format specified in the third value of the Output Format field (9) in the active LND record (language set).

H Use the *H* option to specify 12-hour format. The characters used to indicate AM and PM are identified by the active language set.

Note For more information on language sets, see the "International applications" appendix in the *Reference* manual.

S Use the *S* option to indicate that output includes seconds.

char The *char* option specifies the character that separates hours, minutes, and seconds. The default is a ":" (colon). *Char* can be any character between ASCII characters 33 and 248 inclusive.

Note For more information, see TIME( ) and "ICONV Time (MT)" in this chapter.

Correct use of OCONV Time (MT) Function

```
INTERNAL_TIME = "73680"
CONVERSION    = "MTH"
TIME          = OCONV(INTERNAL_TIME,CONVERSION)
```

The value 08:28PM is saved in TIME.

The following table provides examples of the correct use of the OCONV Time function:

Example	Output
OCONV(43260,"MT")	12:01
OCONV(43260,"MTH")	12:01PM
OCONV(43260,"MTS")	12:01:00
OCONV(43260,"MTHS")	12:01:00PM
OCONV(61200,"MT")	17:00
OCONV(61200,"MTHS,")	05:00:00PM

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 label1[...] [[,labeln[...]]...]
```

index When the ON GOSUB statement is encountered, the current value of *index* is determined, and the program transfers control to the proper subroutine.

If *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.

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 Subroutine within the program. Either numeric or alphanumeric labels are valid.

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
```

(continued)

```
LABEL3:
PRINT SCR.LOC1 : "You pressed a 3 " : SCREEN :
RETURN

LABEL4:
STOP
```

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 *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.

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 l abel 9, and so on.

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 *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"). Use separate OPEN statements to open the DICT and data portions of each file.

filevar After a successful OPEN operation, *filevar* contains information used in subsequent reference to the file.

THEN The *statement(s)* following THEN are executed if a file is opened successfully.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Correct use of OPEN

The following program demonstrates file opening and subsequent processing.

```
DECLARE SUBROUTINE MSG, FSMSG
message = "Enter a record key: "
map     = ""
map     = "RC" ;* case insensitive response mode of MSG
response = ""
values  = ""
file    = "SAMPLE_CUSTOMERS"
not_here = " not available in ":file
```

(continued)

```
OPEN file TO cust_file THEN
   OPEN "DICT ":file TO @DICT ELSE FSMSG();STOP
END ELSE
   FSMSG(); STOP
END
done = 0
LOOP
   MSG(message, map, response, values)
   IF response EQ "END" THEN done = 1
UNTIL done DO
   READ @RECORD FROM cust_file, response THEN
      report = {COMPANY_NAME}:" ":{CITY}:" ":{STATE}
      MSG(report)
   END ELSE
      MSG("%B%":response:not_here)
   END
REPEAT
```

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 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

Note For more information, see OSBWRITE and OSOPEN. For more information on operating system file access, see "Accessing operating system files" 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"
REVREC1<-1> = "Field 2 REC 1"
REVREC1<-1> = "Field 3 REC 1"

REVREC2     = "Field 1 REC 2"
REVREC2<-1> = "Field 2 REC 2"
REVREC2<-1> = "Field 3 REC 2"

FILE = REVREC1 : CRLF : REVREC2
SWAP @FM WITH "," IN FILE
OSBWRITE FILE TO FILE.VAR
IF STATUS() = 0 THEN
   PRINT CRLF : "OSWRITE results follow:"
   PERFORM "PC TYPE OSTEST"
END
OSBREAD NEWFILE FROM FILE.VAR THEN
   PRINT CRLF : "OSREAD results" : CRLF : NEWFILE
END
PRINT CRLF : "OSBREAD results follow"
FILE.LEN = LEN(NEWFILE)
```

(continued)

```
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 OSTEST"
OSCLOSE FILE.VAR
IF NOT( STATUS() ) THEN PRINT CRLF : "OSTEST closed"
OSDELETE "OSTEST"
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

Note For more information, see the OSBREAD, OSCLOSE, and OSOPEN statements. For more information on operating system file access, see "Accessing operating system files" 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.

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

Note For more information, see OPEN and OSOPEN. For more information on operating system file access, see "Accessing operating system files" 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

Note Files to be deleted should not be open. See OSCLOSE. For more information on operating system file access, see "Accessing operating system files" 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.

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.

Note Be aware that OSOPEN is not an R/BASIC file open command. It applies only to operating system files. You therefore cannot apply OSOPEN to a file, then use the R/BASIC READ command to read it.

```
OSOPEN file TO filevar THEN|ELSE statements
```

file The name of the operating system file (including drive or directory page, 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.

Note 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

Note For more information, see OSBREAD, OSCLOSE, and OSBWRITE. For more information on operating system file access, see "Accessing operating system files" 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.

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
```

Using 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.

Note 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

Note For more information on operating system file access, see "Accessing operating system files" 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.

OSWRITE

The OSWRITE statement writes data to a specified operating system file.

```
OSWRITE expression ON | TO file
```

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

Note For more information see OSREAD. For more information on operating system file access, see "Accessing operating system files" 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.

OUT

The OUT statement is used to output a single value to any port.

Note 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.

Caution! The OUT and INP commands require knowledge of direct memory addressing and hardware. Uninformed use may cause serious problems.

Note 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
```

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.

Note 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.

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.

Note 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.

Note For more information, see the PC, ROLLOUT, and SUSPEND commands in the "TCL command reference" chapter of the *Reference* manual, 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.

PERFORM

The PERFORM statement allows a program to send a command to the Command window for execution and then return to the calling program.

```
PERFORM command
```

command A character string that must contain the exact command as it would be typed at the Command window.

When the PERFORM statement is executed, a recursive call to the the Command window 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 INVOICES WITH DATE < '01-01-92'"
PERFORM "EDIT DOS"
```

These two PERFORM statements would select the INVOICES table 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 the Command window:

- 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.

PRINT

The PRINT statement outputs data to the system printer or to the terminal screen.

```
PRINT [expression] [,expression ...] [:]
```

Using 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 the "TCL command reference" chapter of the *Reference* manual.

Note When using the PRINT statement to output to the screen in Advanced Revelation's windowing environment, you should 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 RUN in the "TCL command reference" chapter of the *Reference* manual for more information on (B).

expression Any valid R/BASIC expression.

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.

Note 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.

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 | numeric_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.

numeric_expression You can also use any numeric expression to redirect output. If *numeric_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.

Note 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.

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.

Note For more information, see INPUT.

Correct use of PROMPT

```
PROMPT "="
```

The = (equal 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.

PUT.CURSOR

PUT.CURSOR changes the current video cursor position and shape.

Note 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.

Note 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
```

(continued)

```
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 Must yield 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).

Note 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.

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.

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 read from *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.

Note 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note For more information, see OPEN, READO, READV, MATREAD and WRITE. See also "Dynamic arrays" in "Programming with R/BASIC".

Correct use of READ

```
DECLARE SUBROUTINE FSMSG
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
   FSMSG()
   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 initializes a cursor with a sorted list of customer records. A READNEXT and READ loop 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.

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
```

Use READNEXT 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.

Caution! If you exit a READNEXT loop before exhausting the select list, you must clear the select list using CLEARSELECT. Otherwise, your results may be unpredictable.

variable If a SELECT or GETLIST is performed at the Command window 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* returns the position of *variable* in the multivalued field.

Note For more information about selecting and sorting data, see "Using filters" in the "Introduction to R/LIST" chapter of the *Reference* manual.

THEN The THEN clause is executed if the READNEXT is successful.

Note 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, 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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 ELSE
   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 key in the selected list of keys from the PERFORM is read into @ID by READNEXT.

READNEXT BY

READNEXT BY is an extension to READNEXT that supports multiple select lists. When you use READNEXT, the default cursor (cursor 0) is assumed.

```
READNEXT variable USING cursorvar BY direction ⇨
    THEN|ELSE statements
```

Caution! If you exit a READNEXT loop before exhausting the select list, you must clear the select list using CLEARSELECT. Otherwise, your results may be unpredictable.

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 — Cursor variable. Cursor variables are initialized with a SELECT ... BY statement.

Note For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

direction Indicates the direction in which the keys are read (ascending or descending) and whether the select list is terminating or nonterminating. If the list is non-terminating, the value of STATUS( ) is set to 1 after the last key has been read. *Direction* must be an integer between 0 and 3. You can use alpha codes 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

Note 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.

Note 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 "Accessing Advanced Revelation tables" 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.

Note 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 the *Reference* manual.

Correct use of READNEXT BY

```
DECLARE SUBROUTINE MSG, FSMSG
CURSOR.NO = 0
CLEARSELECT CURSOR.NO
SELECT "CUSTOMER" BY "CITY" SETTING CURSOR_NO ELSE
   MSG("Cannot select the CUSTOMER file!") ; STOP
END
```

(continued)

```
DONE = 0
LOOP
   READNEXT @ID USING CURSOR.NO BY DN ELSE
      FSMSG()
   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 FSMSG("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.

Note 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 Value of the data read from *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.

Note For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

filevar 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note For more information, see OPEN and READ. See also "Dynamic 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.

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 — File variable from previous OPEN statement.

key — Key to the record that is to be read.

field — 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note For more information, see OPEN, READ, MATREAD and WRITE. See also "Dynamic 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.

REMOVE

The REMOVE statement is used to extract substrings, including fields, values, or subvalues, from dynamic arrays using ASCII characters 249 to 255 as delimiters.

```
REMOVE variable FROM string AT column SETTING flag
```

variable, string — Substring and the string from which it is extracted.

column — Indicates the starting position of *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 — 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

Note 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.

REPLACE

The REPLACE function replaces a field, a value, or a subvalue in a dynamic array.

```
REPLACE(expression, field, value, subvalue, new)
```

expression 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.

Note For more information, see < > (angle bracket operators), DELETE, EXTRACT, and INSERT. See also "Dynamic arrays" in the chapter "Programming with R/BASIC".

---

new 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,"")
```

(continued)

```
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.

Note For more information, see CALL, DECLARE, FUNCTION, and SUBROUTINE. See also "External functions and subroutines" in the chapter "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 control to the statement following GOSUB */
RETURN
```

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
```

The following code must be separately compiled and cataloged, before running the previous program, which calls it.

```
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.

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 between 0 and (X-1).

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
```

Using 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 "Accessing Advanced Revelation tables" 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 File variable specified in a previous OPEN statement.

Note 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 "TCL command reference" chapter in the *Reference* manual.

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.

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.

Note 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 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 Specifies the sort order for the record keys to be returned in the cursor.

Note 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 *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 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 — 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Correct use of extended SELECT

```
DECLARE SUBROUTINE MSG, FSMSG
SORT_LIST = "ZIP" : @FM : "COMPANY"
SELECT "CUSTOMER" BY SORT_LIST SETTING CURSOR ELSE
   FSMSG()
   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
      FSMSG()
      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].

SERIAL( )

The SERIAL( ) function returns the Advanced Revelation serial number.

```
SERIAL()
```

Using 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.

SIN

The SIN function calculates the trigonometric sine of an angle.

```
SIN(expression)
```

expression Must yield 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.*

SPACE

The SPACE function generates a string value containing a specified number of blank spaces.

```
SPACE(expression)
```

expression Must yield 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.

SQRT

The SQRT function calculates the square root of a positive number.

```
SQRT(expression)
```

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.

STATUS( )

The STATUS( ) function tests the results of a previous operation to determine its validity.

```
STATUS ()
```

Using 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, OCONV, 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.

STOP

The STOP statement terminates program execution. The statement may appear anywhere in the program.

```
STOP [expression]
```

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.

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 Specifies the string value that is repeated. The expression must be a variable name or character string value.

expression2 Must evaluate to a positive integer. *Expression2* specifies the number of times to repeat *Expression*1. 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.

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 [ (parameter [,parameter] ...) ]
```

Creating subroutines

The subroutine must have been previously compiled and cataloged before it can be called. Use the TCL command SETPROGRAM 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 command 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. The value of *name* is for your use only. Advanced Revelation can only use the name of the compiled record, which could be different from any value of *name* in the subroutine declaration.

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. As with the value of *name*, specific names for arguments have no function outside the code of the subroutine. When a compiled subroutine is called by Advanced Revelation, it handles arguments by their positions in the calling routine.

For more information, see CALL, COMMON, DECLARE, EXPENDABLE, FUNCTION, and RETURN. See also "Subroutines" in the chapter "Programming with R/BASIC", and SETPROGRAM in the chapter "TCL command reference" in the *Reference* manual.

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
```

The following code must be separately compiled and cataloged, before running the preceding program.

```
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.

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 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 = "2]1]2]2"
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 3]2]2.

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.

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
```

Using 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. *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.

Note 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.

Note The SUSPEND statement reads the operating system filename from the system variable @ROLLOUT.FILE. Set @ROLLOUT.FILE using the command ROLLOUT before using SUSPEND.

Caution! 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 the "TCL command reference" chapter of the *Reference* manual. 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.

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.

Note For more information, see CONVERT.

```
SWAP expression1 WITH expression2 IN variable
```

Using 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 *AXYZDEF*.

```
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."

TAN

The TAN function produces the tangent of an angle expressed in degrees.

```
TAN(expression)
```

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"
   END ELSE
      PRINT ANGLE,TAN(ANGLE)
   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.

TIME( )

The TIME( ) function returns the internal time of day.

```
TIME()
```

Using 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 midnight.

TIMEDATE( )

The TIMEDATE( ) function returns a string value expressing the current time and date in external format.

```
TIMEDATE()
```

Using TIMEDATE( )

The TIMEDATE() function returns the time and date in a standard format.

The format for the display of this function is determined by the OCONV Datetime conversion that is part of the active language set. This conversion is specified in value 4 of position 9 (Output Formats) in the language set record format (see the "International applications" appendix in the *Reference* manual).

See "OCONV Datetime (DT)" in this volume for information about specifying appropriate formats. Also, compare 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.

TRANSFER

TRANSFER moves the contents of one variable to another and clears the original variable.

```
TRANSFER variable1 TO variable2
```

Using 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 Source variable. After the execution of a TRANSFER statement, *variable1* is null.

variable2 Target variable. 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
   FSMSG()
   STOP
END
SELECT CUSTOMER_FILE
DONE = 0
```

(continued)

```
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 value */
      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.

TRIM, TRIMF, TRIMB

The TRIM function deletes the extra blank spaces from a character string value or variable.

```
TRIM(expression)
TRIMF(expression)
TRIMB(expression)
```

Using 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
```

Using 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^^^
```

Using 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

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 RESETSYSTEM command or the END command at the debugger prompt.

```
UNLOCK filevar | ALL | CURSOR cursorvar [,key ⇨
    [,locktype]] THEN|ELSE statements
```

filevar File variable from previous LOCK statement.

cursorvar If accessing a cursor, *cursorvar* contains a cursor variable. Cursor variables are initialized with a SELECT ... BY statement.

Note 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 *n*0 is used to indicate an UNLOCK ALL.

THEN The *statement(s)* following THEN are executed if the file or record is unlocked successfully.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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$<D>
LOOP UNTIL LOCKED
   LOCK FILE_CUST, KEY THEN
      LOCKED = TRUE$
   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
   FSMSG()
END
STOP
```

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 | TO filevar | CURSOR cursorvar ,key ⇨
    THEN|ELSE statements
```

expression Evaluates to the record that is to be written to the file.

filevar 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.

Note For more information about cursors, see "Cursors" in the chapter "Programming with R/BASIC". See also SELECT ... BY.

key Specifies the key of the record to be written.

THEN The statement(s) following THEN are executed if the record is written successfully.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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, FSMSG
@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
         FSMSG("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.

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 THEN|ELSE ⇨
    statements
```

expression Must evaluate to the field contents that are to be written to the file.

filevar File variable that has been opened in a previous OPEN statement.

Note The WRITEV statement does not support file access using cursors.

key Key of the record to be written.

field 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.

Note 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 "Accessing Advanced Revelation tables" in the chapter "Programming with R/BASIC".

Note 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 FSMSG
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
         FSMSG()
      END
   END ELSE
      FSMSG("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.

XLATE

The XLATE function is used to retrieve data from a specified field in a file.

```
XLATE(file, key, field, control [,expression])
```

file File from 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 Identifier of record containing 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 subvalue 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 "Accessing Advanced Revelation tables" 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.

```
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 # "")
```

(continued)

```
      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.

6. R/BASIC System Subroutines

Introduction

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

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, the Command window, and so on, considerably eases 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 SYSOBJ file — there is no requirement to load or catalog any of the programs.

As with any subroutine, these system subroutines and functions 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 functions (POP.UP and VIDEO.RW are included here).

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
(etc.)	

You can read the value of these @-variables 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, and restoring the old values before exiting your program.

A note about constants and $INSERT

Advanced Revelation makes use of standardized system constants to provide more readable and maintainable code. These constants generally take the form of English names for flags or numeric values.

For example, in system routines, the word TRUE is normally 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 most cases, the constant name (for example, GENERAL.HELP$) will have a $ (dollar sign) appended to it to help distinguish it from variables in the program.

You also can use these constant definitions. Several items in the SYSINCLUDE 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 invokes them in the program itself:

```
SUBROUTINE TEST
$INSERT SYSINCLUDE,WINDOW.CONSTANTS
$INSERT SYSINCLUDE,WINDOW_COMMON%
*
* (processing here)
*
WC_RESET% = RESET.WINDOW$
RETURN
```

Note that the use of the WINDOW.CONSTANTS values is optional. The next-to-last line in the above program could have been written:

```
RESET = 5
```

and the line:

```
$INSERT SYSINCLUDE, WINDOW.CONSTANTS
```

left out. There is no functional difference between the two methods.

However, we recommend the use of $INSERT statements to include blocks of code to maintain clarity and maintainability.

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. Documentation for additional system subroutines is available in the Developer Series book *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).
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.EXTRACT	Retrieve primary keys based on indexed value(s).
	COLLECT.IXVALS	Retrieve all data values in a particular index.
	INDEX.FLUSH	Update (flush) pending index transactions for one or all indexes.
General	ATTACHTABLE_SUB	Attach volumes and tables.
	CATALYST	Process an Advanced Revelation code and command sequence.
	DETACHTABLE_SUB	Detach volumes and tables.
	FIXLH_SUB	Fix Group Format Errors (GFE).
	PROGRESS	Use status line to graphically show the progress of a process.
	REDUCE	Specify selection (reduction) criteria for a cursor.

(continued)

Process	Program Name	Description
	SETATTACH_SUB	Create an "image" of currently-attached tables and volumes (to be used for a quick-attach process with USE_ATTACH_IMAGE).
	USE-ATTACH_IMAGE	Load an attach "image" created earlier with SET_ATTACH_IMAGE. Speeds the attach process.
	(User-defined Conversions)	Convert data into and out of internal and external format; callable from the R/BASIC ICONV and OCONV functions.
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.
	PROGRESS	Keep user informed about the rate of progress for a process.
	STATUP	Update the Advanced Revelation status line.
	VIDEO.RW	Clear and restore a selected portion of the screen (see also BORDER.UP).
Select lists	DELETELIST_SUB	Delete a list that has been saved.
	GETLIST	Retrieve a list stored in a table.
	SAVELIST	Save a select list from memory to a row in a table.

ATTACHTABLE_SUB

Attaches a table or list of tables to the current application.

```
ATTACHTABLE_SUB( VolumeName, TableList, Options) )
```

Using ATTACHTABLE_SUB

VolumeName — Only one location or volume can be specified. If null, the default system location, as specified in the environment, is presumed.

Be sure to pass this argument as an uppercase value.

TableList — Pass one or more names of tables in the specified volume. If more than one, separate table names with field marks (@FM).

If this value is null, then the entire volume specified in *VolumeName* is attached.

You can preface the DICT or DATA key words to the table names to restrict what gets attached.

Options — Only one option is available: use "A" to have the system announce when a table changes from one volume to another.

For example, if the application currently has a CUSTOMERS table, and this operation results in the attachment of a volume that also has a CUSTOMERS table, the new CUSTOMERS table becomes the currently valid CUSTOMERS table. To alert the user that the contents of that table have changed, the status line (if active) displays a message, or, if the status line is not active, a message appears in a box.

Note: You can also use an "image" of all the tables and volumes you have available in order to speed up the attach process. See SETATTACH and USE_ATTACH_IMAGE in this book, and SETATTACH in the "TCL command reference" in the *Reference* manual.

Correct use of ATTACHTABLE_SUB

```
VolumeName  = "SAMPLE"
TableList   = "CAR_ORDERS":@FM:"CAR_PARTS"
Options     = "A"

CALL ATTACHTABLE_SUB( VolumeName, TableList, Options )
```

This code attaches two tables from the SAMPLE volume. If these tables replace tables already attached of the same name, then this fact will appear on the status line.

BORDER.UP

The BORDER.UP subroutine draws borders around any area of the screen. It shares most of its parameters with VIDEO.RW, and they are often used together.

```
BORDER.UP(left, top, right, bottom, border, attribute)
```

Using BORDER.UP

Use BORDER.UP to put one of the five Advanced Revelation border types on any part of the screen. You may choose to display the border with shadowing as well.

left Pass an argument for the left column of the border area.

top Pass an argument for the top row of the border area.

right Pass an argument for the right column of the border area.

bottom Pass an argument for the bottom row of the border area.

border Either directly assign one of the five Advanced Revelation border types, or use the system default by referencing the border type parameter in one of the environment constants fields (@AW, @PW, etc.).

To add shadowing to your border display, set the sixth field of this argument to true. You could do this with the code:

```
border<6> = 1
```

Shadowing appears along the bottom and right borders. This position cannot be changed. Shadowing will only be activated when the system environment flag (@ENVIRON.SET<76>) is true.

attribute Pass a value to specify the video attribute of the border. This procedure is different for color and monochrome.

Values returned

None.

Correct use of BORDER.UP

The following code clears an area on the screen, substitutes a particular background for the default screen, displays a border in the lower part of the screen, and then displays another borde, this time with shadowing.

```
DECLARE SUBROUTINE BORDER.UP, MSG, DELAY
DECLARE FUNCTION VIDEO.RW

EQUATE true$      TO 1
EQUATE false$     TO 0
EQUATE shadowon$  TO 6

verrors     = ""
verrors<1>  = "Out of memory during read attempt"
verrors<2>  = "Display is not memory-mapped"
verrors<3>  = "Col or Row is out of range"
verrors<4>  = "Left col is greater than right col, or"
verrors<4,-1>  = " top row is greater than bottom row"
verrors<5>  = "MODE is not 'C', 'R', 'W'"
verrors<6>  = "Size of saved image is less than area ⇨
   to be restored"

@ENVIRON.SET<76> = 1

left = 10
save_left = left
top = 5
save_top = top
right = 70
save_right = right
bottom = 20
save_bottom = bottom
background = CHAR(32)    ;* a space
attribute = \1F\         ;* white text on blue
operator = "R"

GOSUB call_video.rw
save_image = image
operator = "C"
image = background:attribute
GOSUB call_video.rw
```

(continued)

```
left = 15
top = 14
right = 40
bottom = 19
border = 2
border<shadowon$> = false$
attr  = CHAR(27):"C8?"  ;* highlight blink

BORDER.UP(left, top, right, bottom, border, attr)
DELAY(5)
TRANSFER save_left TO left
TRANSFER save_top TO top
TRANSFER save_right TO right
TRANSFER save_bottom TO bottom
TRANSFER save_image TO image

operator = "W"
GOSUB call_video.rw
STOP

call_video.rw:
error = VIDEO.RW(left, top, right, bottom, ⇨
   operator, image)
IF error THEN
   MSG("Aborting...|":verrors<error>)
   STOP
END
RETURN
```

BTREE.EXTRACT

The BTREE.EXTRACT subroutine will search one or more Btree indexes for data that matches the search criteria passed to BTREE.EXTRACT. It returns the keys of records having matching data.

```
BTREE.EXTRACT(srch_strng, file, dictvar, keys, option,⇨
   flag)
```

Using BTREE.EXTRACT

Access to Advanced Revelation Btree indexes is normally handled transparently at the time a SELECT is generated or when you do a [Ctrl-F10] search from a window. You can also have the system do direct lookups into Btree indexes by using a CATALYST "B" call, but this requires user interaction. For more direct access, use BTREE.EXTRACT.

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 you 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 via an Environmental Bond.

BTREE.EXTRACT also allows you to provide your own routines for preprocessing search data, as well as for your own search algorithm. See "User Index" following the sample code below.

You must open the associated dictionary file before calling BTREE.EXTRACT.

srch_strng The basic unit of *srch_strng* is a search field. This field consists of the name of an indexed field together with search values to be located within that field:

```
search_field = indexed_field:@VM:data1:@VM:data2:@FM
```

A Btree index must have been applied to the specified record field, or an error is generated. Within the search field, search values are delimited from the indexed field name and each other by a value mark (@VM). Within the search field, search values are located based on an OR relation: a record key is returned when any of the values is found within the specified field.

The implied OR relation for the @VM-delimited search values may be forced to an AND relation by prefixing any desired search value with "&" (ampersand).

If you want to search by more than one criterion, include additional search fields in *srch_strng*. Each search field is delimited from another by a field mark (@FM), and multiple search fields imply an AND relation: conditions must be satisfied for all search fields before any key is returned. The AND relation may be overridden for individual search values by prefixing each with a semicolon (;).

Srch_strng must end with a field mark.

Returning more than 64K of keys

If the index you are searching is particularly large, you may end up with more than 64K of keys that match your search criteria. In that case, you can call BTREE.EXTRACT multiple times, once to search, and subsequent times to return lists of keys beyond the first return values.

To search large indexes, you must indicate that you are willing to accept more than 64K of keys. BTREE.EXTRACT will then return to you a "set handle", which you can use later to query for further keys.

Search examples

The search values in *srch_strng* are substrings, and as such, you can modify the search relations according to Query substring search characters (see the chapter "Using Query" in *Accessing Data* in the main documentation). For example,

```
srch_strng = "COMPANY_NAME":@VM:"#TRUST INSURANCE":@FM
```

will find all values of the indexed field COMPANY_NAME that are *not* "TRUST INSURANCE".

BTREE.EXTRACT supports a search operator in addition to those used with Query: *between* ("~" (tilde)). *Between* differs from *range* in that it is not inclusive. For example, the following search string will return record keys for all records having zip codes between, but not including, 98100 and 98111:

```
srch_strng = "ZIP":@VM:"98100~98111":@FM
```

Searching by AND

As noted above, multiple search criteria are linked with an implicit AND. The TCL-level statement WITH STATE = WA AND CITY = SEATTLE is performed in BTREE.EXTRACT by the code:

```
"STATE":@VM:"WA":@FM:"CITY":@VM:"SEATTLE":@FM
```

Searching by OR

To change the implicit AND to an OR, insert a semicolon (;) before each search value in the second (and subsequent) search field(s). The statement WITH STATE = WA OR CITY = SEATTLE is performed in BTREE.EXTRACT by the code:

```
"STATE":@VM:"WA":@FM:"CITY":@VM:";SEATTLE":@FM
```

file Use *file* to pass the name of the file to be searched.

dictvar Use *dictvar* to pass the file handle for the dictionary of the specified file.

keys Record keys for the records that satisfy the criteria in *srch_strng* are returned in *keys*. Multiple keys are delimited with value marks.

option These values for *option* are possible:

Option	Meaning
null	All messages will display.
S	All informational messages will be suppressed.

If you are willing to accept more than 64K of keys, *option* is treated as a dynamic array with these values:

Option	Meaning
new query	
<2>	1 to indicate start of new query
<3>	Passed as null (returns with set handle if result set > 64K)
more keys	1 to indicate continuation of earlier query.
<2>	1 to indicate start of new query.
<3>	set handle returned previously in *option*<3>. If last group of keys, *flag* returns 0.
stop query	
<2>	-1 to abort query
<3>	set handle of query being aborted

flag Error codes are returned in *flag*.

Values returned

After execution, *keys* contains a list of keys matching the search criteria. *Flag* returns one of several possible values, depending on the success of the search process:

Value	Meaning
0	The search was successful. All possible (or the last set of) keys were returned in *keys*.
>0	The value of *flag* represents the total number of keys returned in *keys*.
-1	The search failed, for reasons other than no keys found.
n	(positive integer value > 1) If more than 64K of keys is found, *flag* contains a count of the number being returned this time.

Correct use of BTREE.EXTRACT

The following code opens a file, then prompts for the fields to search and for values to locate within those fields. BTREE.EXTRACT is used to obtain record keys, which are displayed upon completion.

```
DECLARE SUBROUTINE MSG, INPUT.CHAR,
DECLARE SUBROUTINE FSMSG,  BTREE.EXTRACT
EQUATE false$  TO 0
EQUATE true$   TO 1

index = "COMPANY_NAME"
error = false$

file = "SAMPLE_CUSTOMERS"

OPEN "DICT.":file TO @DICT ELSE
   FSMSG()
   STOP
END

LOOP
   error = false$
   MSG("Enter name of field to search: ","RC",index,"")
   IF index THEN
      READ field_name FROM @DICT, index ELSE
         FSMSG()
         error = true$
      END
   END ELSE STOP
WHILE error REPEAT
value   = "Security Insurance"
value_text = "Enter values to search for, separated ⇨
   by commas|"
MSG(value_text,"RC",value,"")
IF value THEN
   CONVERT "," TO @VM IN value
   srch = index:@VM:value:@FM
   BTREE.EXTRACT(srch, file, @DICT, keys, option, flag)
   IF flag THEN
      IF flag > 0 THEN
         msg_no = 4
      END ELSE
         msg_no = 3
      END   ;* if flag > 0
   END ELSE
```

(continued)

```
      IF keys THEN
         msg_no = 1
      END ELSE
         msg_no = 2
      END   ;* if keys
   END      ;* if flag

   messages    = ""
   messages<1>   = "Successful search: keys returned"
   messages<1,2> = are:|":keys
   messages<2>   = "No matching keys were found."
   messages<3>   = "Search failed."
   messages<4>   = "Search was successful, "
   messages<4,2> = "but list  > 64K."
   messages<4,3>  = "Returned keys are:|":keys
   MSG(messages<msg_no>)
END   ;* if value
STOP
```

A second example shows how to process more than 64K of keys. Some of the arguments to BTREE.EXTRACT are not explicitly set in this code fragment for the sake of brevity.

```
OPTION = ''
OPTION<1> = "S" ; * suppress messages
OPTION<2> = 1 ; * begin new query
OPTION<3> = '' ; * empty set handle
SEARCH = "CUSTOMER":@VM:"SMITH" : @FM
LOOP
   BTREE.EXTRACT(SEARCH,FILE,@DICT,KEYS,OPTION,FLAG)
WHILE FLAG > -1        ; * error condition
   GOSUB PROCESS_KEYS ; * local sub not shown here
WHILE FLAG      ; * if flag < 1 then done
REPEAT          ; * more keys to come
IF FLAG < 0 THEN
   GOSUB PROCESS_ERROR ; * error; not shown here
END
```

CATALYST

The subroutine CATALYST processes all Advanced Revelation code and command operations.

```
CATALYST(code, command)
```

Using CATALYST

Use of CATALYST requires knowledge of specific code and command operations. Each operation is different, but use of CATALYST itself is quite straightforward.

Note For more information on codes and commands, see "Using codes and commands" in the *User's Guide*.

Code and command operations are a kind of shorthand: most simply call other Advanced Revelation tools. CATALYST provides a uniform interface to these tools. Use CATALYST in R/BASIC programming to access such standard system-provided functions as window or popup processes.

In addition to the CATALYST parameters *code* and *command*, there are also subcodes, which are modifications of the code operations. Subcodes can be either code-dependent or code-independent. For example, the code X has a subcode C (clear screen). This subcode applies only to code X. But the code-independent subcode R applies to all code and command operations.

Note Code-dependent subcodes are documented below, directly under the relevant code description. Code-independent subcodes, which can apply to all codes, are documented following the section on code descriptions.

Arguments for *command* supply whatever data is needed to execute the operation specified in *code*. Commands can range from null to complex, depending upon the code or operation involved.

Code and Command

Code	Use
!	Returns date, time, time & date, AREV serial #.
@	Executes a captured keystroke set.
{	Evaluates a symbolic field in the current dictionary.
A	Returns to the last anchored menu.
B	Uses a Btree index to look up the record key for known value.
C	Compiles and executes R/BASIC source code.
E	Executes a TCL command.
F	Executes a function key.
H	Executes one of several help-related processes.
K	Executes a series of keystrokes.
M	Displays a menu.
O	Opens a data file within a window.
P	Displays a popup.

(cont.)

Code	Use
S	Calls a cataloged R/BASIC subroutine.
V	Displays a popup of indexed field values.
W	Displays a window.
X	Performs a TCL command.

! Use the ! code when you want to return the date, time, time and date, or the Advanced Revelation serial number.

Code	Command	Example
!	"DATE"	CATALYST("!","DATE")
!	"TIMEDATE"	CATALYST("!","TIMEDATE")
!	"TIME"	CATALYST("!","TIME")
!	"SERIAL"	CATALYST("!","SERIAL")

All return values are returned in @ANS (and therefore to the prompt, if called from within a window).

@ Use the @ code when you want to execute a captured keystroke set.

Code	Command	Example
@	capture_record	CATALYST("@","CAPTURE_RECORD")

An argument for *capture_record* is required and must be the name of the record in the CAPTURED file that is to be executed.

{

Use the { code to evaluate a field in the current dictionary. This code will run any R/BASIC code in the specified dictionary record and return the result in @ANS.

Code	Command	Example
{	dict_record	CATALYST("{","INV_TOTAL")

The argument for *dict_record* must be a record in the current dictionary. The R/BASIC routine CALCULATE puts the result in @ANS.

The dictionary must be opened to the system variable @DICT during this code and command procedure, and a file record must be in @RECORD.

A Use the A code to return to the last anchored menu, skipping intermediate menus.

Code	Command	Example
A	none	CATALYST("A","")

No command is processed for this code.

B Use the **B** code to use a Btree index to look up the record key for a known data value. This function displays a Shared Index Information popup of the indexed field(s) specified in *command* and searches the Btree index, returning a single selection popup (configurable) of the record keys and display fields. Any user selections from the popup are returned in @ANS.

Code	Command	Example
B	four_part	CATALYST("B","INV*DT*ID1,ID2,ID3*MULTI")

There must be a four-part argument for *command*, where parts are separated from the other by an asterisk. The structure of this parameter is:

```
file*indexed_field*display_field*mode
```

where *file* is the name of an indexed file, *indexed_field* is the name of one or more comma-delimited fields to be searched in the specified file, *display_field* (optional) is the name of one or more comma-delimited field names that will be displayed in the popup that results from the search, and *mode* is either a literal (KEY or MULTI) or null (single selection). KEY returns keys in a browse list, while MULTI returns keys that are @VM-delimited.

The example above opens the INV file, asks the user for one or more dates, searches the index for those dates, and then displays the results of the search in a 3-column multiple selection popup in which each column displays data from its respective display field, and where DT data is not included in the display.

C Use the **C** code to compile and execute the R/BASIC source code specified in *command*. Separate lines of source code with an "&" (ampersand). Total code may extend to 34K for DOS and 64K under OS/2.

Code	Command	Example
C	source	CATALYST("C", "@ANS = SUM({AMOUNT})")
CV	object	CATALYST("CV", object_record)

If you use the V subcode, the argument for *command* may be compiled object code. For example, the command SUM({AMOUNT}) will sum the multivalued field AMOUNT and return the total in @ANS.

D The **D** code executes a RETURN.

E Use the **E** code to execute the specified TCL command.

Code	Command	Example
E	TCL_command	CATALYST("E","RENAMEFILE BP APP_BP")

The argument for *command* must be a valid TCL command. No active list of records will be available to that command. CATALYST runs the R/BASIC EXECUTE statement in *command*. See the **X** code below for more information.

F Use the **F** code to execute a function key.

Code	Command	Example
F	function_key	CATALYST("F", "F2")

Arguments for *function_key* are codes to stand for the various function keys. Examples: F1 is [F1], SF1 is [Shift-F1], CF1 is [Ctrl-F1], and AF1 is [Alt-F1].

H Use the series of **H** codes to display and process help. Each subcode modifies the basic help display process.

The **H** codes and commands are intended to be run within window processes. Many cannot be run outside a window environment because required system variables are not available or have not been initialized.

Arguments for *command* for the **H** series of codes depend upon the subcode. Without a subcode, the argument must be a number. That number is concatenated by CATALYST to the string "AW*template_name*" to make a valid record key for access to the SYSHELP or HELP file. *Template_name* is derived from the current window (the WINDOW_COMMON% variable WC_TEMPLATE%).

H Use the **H** code to display help that is stored in the SYSHELP or the HELP file. The value of *command* is a number that is placed at the end of the record key. A full record key has the format: AW**template_name*number*.

Append a "#" (pound sign) to any of the **H** codes to protect the help message from editing.

HD Use a **D** to display help from the *description* field in a dictionary record. An argument for *command* is not necessary, in which case the current prompt name (field 2 of the AREV.COMMON variable SI) is used as the key to the dictionary record. In either case, it is field 14 of the dictionary record that is displayed.

Prior to an **HD** CATALYST call, the dictionary file must have been opened to SRC.DICT.

HE Use an **E** to explicitly allow editing of a help message. Otherwise, this code is the same as **H**.

HF Use an **F** to read the help message from the file and/or key and/or field specified in *command*. Contents of *command* is a string, with commas separating *filename* from *record_key* and from *field_number* within *record_key*. Unless you set protection (with a "#" subcode), a user can edit this message.

Use the **F** subcode to read a record from HELP, SYSHELP, or another file, or to read a single field (for example, field 14, the description field, from a dictionary record). There are three ways to specify the argument for *command* to reflect those options:

```
filename, record_key, field_number
filename, record_key
record_key
```

The following command displays the contents of field 4 of the record SPECIAL_RECORD from the user-supplied help file SPECIAL_HELP:

```
CATALYST("HF", "SPECIAL_HELP, SPECIAL_RECORD, 4")
```

HFT Use an **FT** subcode to display a help message from the SYSHELP file. When the [F2] key is pressed while reading that help message, the user sees a browse list of related topics. The initial help message is a record key in SYSHELP.

The comma-delimited second part of the argument can also be a record in SYSHELP, but one in which each field is the name of another record key in SYSHELP. That record provides the list of keys that make up the browse list.

As shown in the code example below, instead of passing the key of a list record as the comma-delimited second part of the HFT argument, you may simply pass the @FM-delimited record keys as the comma-delimited second part of the HFT argument.

```
CATALYST("HFT","INV_HLP,ENT_HLP":@FM:"TAX_HLP")
```

This command displays the contents of a help record in the SYSHELP file called INV_HLP, then makes available a browse list of related help records: ENT_HLP and TAX_HLP.

Note The message window is sized in accordance with the primary message specified in *command*. This means that the window will be no larger for additional messages from the browse list than for that primary message. Try to ensure that the primary message provides a reasonable window size.

HFW Use an **FW** subcode to substitute the current window prompt literal (field 5 of the AREV.COMMON variable W(W.CNT)) for the default help window title. This forces the title of the help display to read "Detail Help for...[the current window prompt literal]".

HL Use an **L** subcode to display the literal help message specified in *command*.

HM Use an **M** subcode to display a help message and request a response. The response is returned in @ANS by this routine.

Command is either literal text or a system message stored in MESSAGES or SYS.MESSAGES, referred to by record key. CATALYST first checks MESSAGES then SYS.MESSAGES for a record key corresponding to *command*. If no key is found, CATALYST will assume that you have passed literal text.

The following command will display the "HELP_MESSAGE" record in either system message file:

```
CATALYST("HM", "HELP_MESSAGE")
```

HMD Use an **MD** subcode to display literal text or a message from the MESSAGES file until a key is pressed. See **HM** above.

HT Use a **T** subcode to call MSG and display a message for a fixed duration. *Command* must contain the literal message. Append a "~" (tilde) and a number to the end of your literal string to specify display duration. Otherwise, the system default value (@ENVIRON.SET) is used (2 seconds by default).

Code	Command	Example
H	message_no	CATALYST("H","SQL_UPDATE")
HD	dict_descrip	CATALYST("HD","")
HE	message_no	CATALYST("HE","SQL_UPDATE")
HF	"*file,key,field*"	CATALYST("HF", "ACCT_HELP, CHECKS,6")
HFT	module_key	CATALYST("HFT","F.KEY,F.KEY_LIST")
HFW	"*file,key,field*"	CATALYST("HFW","ACCT_HLP,CHECKS,6")
HL	literal	CATALYST("HL","Here is some help: ")
HM	key or literal	CATALYST("HM","B213")
HT	timed HM	CATALYST("HT","Here's more help: ~10")

I The **I** code executes a RETURN.

K Use the **K** code to execute the series of keystrokes specified in *command*. You can create those keystrokes as you type the contents of *command* simply by toggling the keystroke translator in the Advanced Revelation editor: [Ctrl-] (Ctrl-Hyphen). Instead of performing their defined operations, function and editing keys display literal values. For example, when you press the [F6] function key, the characters "{F6}" appear instead of [F6] taking immediate effect. The K code then translates the literal characters into the actual function of the specified key.

Code	Command	Example
K	func_keys	CATALYST("K","{F5}{SPACE}Hey{SPACE}there!")

The argument for *func_keys* may be a string generated using the keystroke translator key or one typed directly, where function and editing key names are surrounded by braces ({}).

L The **L** code is similar to the **S** code in running cataloged subroutines, but lacks access to window registers. Use the *S* code for all access to subroutines.

M Use the **M** code to display a menu. *Command* contains the name of the menu.

Code	Command	Example
M	menu_record	CATALYST("M", "CUST.SERVE")

The argument for *menu_record* must be the name of the menu to be displayed. CATALYST looks for the menu first in the MENUS file, then in the SYS.MENUS file.

O Use the **O** code to open a data file within a window when that data file is other than the default.

Caution! Use the **O** code *only* for post-prompts in a window process. Any other use will break execution of routines, because certain system variables (IS, SRC.FILE) have not been initialized.

The AREV.COMMON file variable for the data file used within a window is SRC.FILE.

P Use the **P** code to display a popup. There are several different ways to specify the popup to be displayed.

Code	Command	Example
P	file_record	CATALYST("P","CUSTOM_POPUPS*TEST")
P	sys_variable	CATALYST("P","@FILE:15:20:5")
P	window_reg_var	CATALYST("P","@REGISTER1")

Arguments for *command* can be either a user-defined popup name (in the format *filename*recordname*), or one of several system variables or user-defined window registers.

User-defined popups

For user-defined popups, *filename* is the file where the popup template is located, and *recordname* is the record key of the desired popup.

For example, CATALYST("P", "CUSTOM_POPUPS*TEST") runs the TEST popup from the CUSTOM_POPUPS file.

System variables and popups

@ID, @FILE, and @VOLUME produce single-selection popups, while @IDS, @FILES, and @VOLUMES produce multiple- selection popups.

For @ID, @FILE, and @VOLUME, you can set the display size and location of the popup. Use a literal string "*x*:*y*:*z*" after the variable name, where *x* is the width of the popup display, *y* is the starting column, and *z* is the starting row.

For example, CATALYST("P", "@FILE:15:20:5") would display a list of attached files in a popup 15 characters wide, starting at column 20 and row 5.

@ID and @IDS require the AREV.COMMON variable SRC.FILE to contain a datafile file variable.

@TABLE and @TABLES are identical to @FILE and @FILES, with the exception that both exclude system tables and tables without dictionaries.

Window registers with popups

You can use the @REGISTER*n* option to dynamically provide data to a prompt by altering the data in an existing record or by replacing at will the file that provides the popup data.

You must place the following information into one or more window registers:

```
record_structure:char(247):mode_info:char(247):filename
```

where *record_structure* is a valid popup record, *mode_info* is data for the fourth argument of POPUP, and *filename* is the third argument of POPUP. This data is dependent on the mode that has been chosen for the popup (table, row, or field).

Mode_info and *filename* must be delimited from the popup record by a CHAR(247).

You may also use the @REGISTER*n* option to change the input filename and/or record keys for a system popup.

S Use an S code to call a cataloged R/BASIC subroutine.

Code	Command	Example
S	subroutine	CATALYST("S","USER_SUB,10")
S	subroutine	CATALYST("S","USER_SUB,<1>")

An argument for *command* must contain the name of a cataloged subroutine. In addition, if arguments are to be passed to the called subroutine, they must be separated from the subroutine name by a comma, but the entire argument must be passed as a literal. For example, the code

```
CATALYST("S","USER_SUB,10")
```

will invoke the system subroutine USER_SUB with an argument of 10.

You can only pass one argument using this syntax, and that argument cannot contain a comma. If your routine presumes multiple parameters, use a different delimiter (such as an asterisk), and code your subroutine logic like this:

```
SUBROUTINE SAMPLE(in_value)
CONVERT "*" TO @FM IN in_value
arg1 = in_value<1>
arg2 = in_value<1>
arg3 = in_value<1>
```

See the discussion of the command for the **B** code for an example of when CATALYST uses this type of logic.

The argument can be passed as a window register number, which is specified as *<n>*, where *n* is the window register number. From within a window environment, the following code will provide the argument to USER_SUB from a window register:

```
CATALYST("S", "USER_SUB,<1>")
```

V Use a **V** code to display a popup of the indexed field values specified in *command*. Any user selections will be returned in @ANS.

Code	Command	Example
V	*"file,field,sel"*	CATALYST("V","SAMPLE_INVOICES,DATE,M")

The argument for *command* must be a string of comma-delimited values of the structure:

```
filename, fieldname, select
```

where *filename* is an indexed file, *fieldname* is an indexed field, and *select* is either "M" (=multiple selections) or "S" (=single selections).

The previous example displays a multiple selection of data in the DATE field of the SAMPLE_INVOICES file, returning multiple selections.

W Use a **W** code to display a window. If you do not include a value in *command*, the system will ask for the name of a window.

Code	Command	Example
W	window_name	CATALYST("W", "CUSTOMER TABLE.ON")

Any valid window command line option (such as TABLE.ON) may be included, separated by a space from the window name.

Window names are looked up in the WINDOWS file.

X Use an **X** code to execute a TCL command. This code invokes the R/BASIC PERFORM command.

The argument for *command* must contain a TCL command to be performed. CATALYST runs the R/BASIC PERFORM statement on *command*. See "PERFORM versus EXECUTE" under the R/BASIC PERFORM command for more details on the difference between the CATALYST E and X codes.

XC Add the **C** subcode to the **X** code to clear the screen before running the TCL command.

Code	Command	Example
X	R/BASIC_command	CATALYST("X","SELECT CUST BY ST")
XC	R/BASIC_command	CATALYST("XC","LIST CUSTOMERS")

Subcodes

The *code* parameter can accept several tokens other than the code arguments themselves.

Code & Subcode	Command example
S;S;P	PROGRAM1;PROGRAM2; POPUPS*INSURANCE.CODES
@SP	SIMULATE.INPUT@
SR	INVOICE.RPT,<1>
SX	AUDIT.UPDATE

; Multiple code and command sets can be passed if separated by semicolons. If specifying three sets, for example, pass all three code arguments together, separated by semicolons. Then structure the command argument similarly, using semicolons to hold a position even if the code argument does not require a command argument.

= Use the "=" subcode to load @DATA, the keyboard buffer, with the contents of @ANS, which can have been set by the !, {, B, HM, P, and V and codes. A CHAR(13) value (carriage return) is automatically added to the string in @ANS.

: Use the ":" subcode to load @DATA with the contents of @ANS. Unlike the "=" subcode, a carriage return is not added.

M Use the **M** subcode to keep new values from overwriting the data in @ANS. Instead, new values are merged (appended) to the current data in @ANS. This can be useful in a window environment.

R Use the **R** subcode to save the screen image before the code and command procedure is executed. This option also restores the screen image after execution of the command.

P Use the **P** subcode to suspend the playback of captured keystrokes before executing the specified code and command procedure. See "The Code and Command Options" in the "Using Codes and Commands" chapter of *Developing an Application* for information on adding codes and commands to captured keystroke sets.

X Use the **X** subcode to specify that a post-prompt code and command procedure will run only when data at the prompt changes. You might use the X suffix with the S code and command to run a post-prompt procedure that writes an entry to an audit file each time the value at a prompt changes.

COLLECT.IXVALS

COLLECT.IXVALS is a subroutine used to return a list of all the data values in a particular Btree index.

```
COLLECT.IXVALS(file_name, index_name)
```

Using COLLECT.IXVALS

In returning all data values, COLLECT.IXVALS 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.

file_name Use *file_name* to pass the name of the data file for which the index lookup is being done.

index_name Use *index_name* to pass 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.

Correct use of COLLECT.IXVALS

```
DECLARE SUBROUTINE MSG, INPUT.CHAR, FSMSG,
COLLECT.IXVALS

/*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 FSMSG(); STOP
* prompt for file name
LOOP
   error = 0
   file_name = "SAMPLE_CUSTOMERS"
   TEXT = "Enter file name to fetch index for"
   MSG(TEXT, "RC", file_name, "")
```

(continued)

```
   IF file_name THEN
      READ file.info FROM files.file, file_name ELSE
         MSG(file_name:" not a valid file!")
         error = 1
      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 = "COMPANY_NAME"
   TEXT = "Enter name of index to display"
   MSG(TEXT, "RC", 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 = ""
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
```

DELETELIST_SUB

Deletes a list from a specified table.

```
DELETELIST_SUB( DeleteTable, ListName)
```

Using DELETELIST_SUB

Use DELETELIST_SUB to remove lists that you have stored with SAVELIST_SUB.

DeleteTable Table from which the list is to be deleted (usually LISTS)

ListName Row key for list to be deleted.

Correct use of DELETELIST_SUB

```
DeleteTable = "APP_LISTS"
ListName    = "MICHAEL"

DELETELIST_SUB( DeleteTable, ListName)
```

This code removes the MICHAEL list from the APP_LISTS table.

DETACHTABLE_SUB

Detaches a volume or any tables within a volume, from the current application.

```
DETACHTABLE_SUB ( VolumeName, TableList)
```

VolumeName Name of volume having the tables in *TableList*. If *VolumeName* is null, then the system default volume is used. If a volume name is specified, no *TableList* can be passed.

TableList If no *VolumeName* is passed,. the list of tables to detach. Separate the names with a field mark (@FM). You can specify the DICT or DATA key words, to restrict what gets detached.

Correct use of DETACHTABLE_SUB

```
VolumeName   = ""
TableList    = "CAR_ORDERS":@FM:"CAR_PARTS"
DETACHTABLE_SUB( VolumeName, TableList)
```

This code detaches two tables.

FIXLH_SUB

Fixes Group Format Errors.

```
FIXLH_SUB ( TableName, FixGroup, Options )
```

Using FIXLH_SUB

FIXLH_SUB is the R/BASIC equivalent of the FIXLH TCL command.

TableName Any attached table.

FixGroup Specifies the group or range of groups to be fixed. If an ending value is passed, separate it from the beginning value with a field mark.

Pass a value in one of the following formats:

GroupNumber	fix one specific group, or, if null or "*", fix the entire table
GroupNumber:@FM:"*"	fix the table, from the specified group to the end of the table
BeginningNumber:@FM:EndingNumber	fix the table, from the beginning group through the ending group

The end of the table is determined by looking at the modulo kept in the table header. If the modulo is wrong, then not all groups will be visited.

On the other hand, if the modulo is wrong, you've got a GFE in the first group

options

S Suppress output to the monitor

C Compress the file (remove unused OV frames), after the fix

Correct use of FIXLH_SUB

```
FIXLH_SUB( "MYTABLE", "", "" )
```

All groups in MYTABLE are fixed.

```
FIXLH_SUB( "MYTABLE", "*")
```

All groups in MYTABLE are fixed.

```
FIXLH_SUB( "MYTABLE", 100 )
```

Only group 100 is fixed.

```
FIXLH_SUB( "MYTABLE", 100:@FM:150, "C")
```

Groups 100 through 150 are fixed, and the table is compressed.

```
FIXLH_SUB( "MYTABLE", 150:@FM:"*", "")
```

All groups, starting with group 150 and going to the end of the table, are fixed.

FSMSG

Use the subroutine FSMSG to report file system errors. Each file system error has an associated message in the SYS.MESSAGES file, which will display with a call to FSMSG.

```
FSMSG([user_text, user_params])
```

Using FSMSG

Many file system functions set an error code in @FILE.ERROR<1>. FSMSG looks for a code in @FILE.ERROR, then displays the associated message text (@FILE.ERROR<2>). You may add to that text by including arguments with your call to FSMSG. Your message text appears above that provided in SYS.MESSAGES. In addition, if your message text includes variables, a second argument can provide values for these variables.

If there is no error code in @FILE.ERROR and FSMSG is called anyway, it will report "Null I/O Error".

When FSMSG is called without arguments and without using the CALL statement, it must include trailing parentheses. For example, CALL FSMSG and FSMSG() are equivalent, but FSMSG alone is not allowed.

user_text With a *user_text* argument, you can add text to the message requested in @FILE.ERROR. The same rules for text and replaceable parameters apply to this parameter as for passing text using MSG.

user_params If you are using replaceable parameters (%0%, %1%, etc.) in *user_text*, pass their arguments in *user_params*. This is where you provide the values that replace those variables at display time. Delimit multiple arguments for replaceable parameters with @VM.

Note This parameter accepts the same arguments as the *values* parameter of MSG (see MSG for more details). For example, you can add date and time information to your file system messages with the %D% and %T% parameters.

Correct use of FSMSG

The following code provides two attempts to open a non-existent file. In the first, only system-generated messages are displayed. For the second attempt, user-provided text is added to the system message.

```
DECLARE SUBROUTINE FSMSG

OPEN "NO_FILE" TO filevar ELSE
   FSMSG()
END

OPEN "NO_FILE" TO filevar ELSE
   TEXT = "for message %0%, errant file is %1%"
   FSMSG(TEXT, "NO_FILE")
END
```

GETLIST_SUB

Retrieves a list of keys from the specified table, and activates it as a select list.

```
GETLIST_SUB( TableName, RecordName, Options )
```

TableName Table from which to retrieve the list of keys. If null, the default value is the LISTS table.

RecordName Row identifier (key) of the key list. An argument for this parameter is required.

Options The "O" option allows overwriting an existing active select list. Otherwise, this routine will not overwrite any existing active list.

Correct use of GETLIST_SUB

```
GETLIST_SUB( "", "QUERY101", "O" )
```

Retrieves the keys stored in the QUERY101 row of the LISTS table, turning them into an active list.

INDEX.FLUSH

INDEX.FLUSH is a subroutine that forces a Btree index update, when required.

```
INDEX.FLUSH(file_spec, index)
```

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.

file_spec The source data file 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 following code demonstrates index flushing in three situations: when a file specification and indexed field are provided, when a file is specified without declaring a field, and when INDEX.FLUSH is to update all indexes for all files.

```
DECLARE SUBROUTINE INDEX.FLUSH, MSG, FSMSG

file = "SAMPLE_INVOICES"
account = "GLOBAL"
volume = "DATAVOL"
```

(continued)

```
OPEN "VOLUMES" TO volume_var THEN
   READ volume_rec FROM volume_var, volume THEN
      media = volume_rec<1>
   END ELSE
      FSMSG()
   END
END
file_spec = file:"*":account:"*":media
indexed_fld = "DATE"
* update the specific index
INDEX.FLUSH(file_spec, indexed_fld)
* update all indexed fields in the specified file
INDEX.FLUSH(file_spec, "")
* update all indexed fields in all files
INDEX.FLUSH("", "")
STOP
```

INIT.VIEW

The subroutine 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 by using the PRINT statement, the data simply displays over what is currently on the screen. If the data is too wide for the screen, it wraps to the next line. In addition, you can only page down, not up.

To get around these limitations, print to a View window. The View window captures information printed with the PRINT statement and formats it into a scrollable, pageable window. You can create a View window for your reports by calling the system subroutine INIT.VIEW.

INIT.VIEW is used in conjunction with the system variable @VIEW.MODE. Before calling INIT.VIEW, you must initialize the View window mode by setting this variable to true.

Call INIT.VIEW only from a main program. It should not be used in an external subroutine.

If you use the R/BASIC PERFORM SELECT statement together with a call to INIT.VIEW, you must perform the SELECT statement before calling INIT.VIEW.

INIT.VIEW should not be used with reports that are being sent to a printer.

Note For more information about the View window, see the chapter "Introduction to queries in Advanced Revelation" in the *User's Guide*.

Although under R/LIST you may use the [Alt-V] and [Alt-P] keys to affect screen and printer output, respectively, those options are not valid when displaying to a View window via INIT.VIEW.

INIT.VIEW is designed to be used with the paging capabilities offered by the R/BASIC statements HEADING and FOOTING, but does not require their use. 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.

width An argument for this parameter sets the width in columns for the View window. This should represent the greatest of the following 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. Data too wide for the View window will wrap.

Values returned

None. Error messages are generated during execution.

Correct use of INIT.VIEW

The following code prepares a report display for data in the VOC file. The width of the View window is determined according to the number of columns present in the display. Each field in each record is afforded one display column, but only three fields are displayed.

```
DECLARE SUBROUTINE MSG, INIT.VIEW, FSMSG
* 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
col_count = COUNT(col_head, @FM)
view_len = SUM(col_len) + col_count
* initialize heading and footing
head = "Sample report for VOC file"
foot = "'L'Page 'P'"
```

(continued)

```
* calculate length of heading/footing
head_len = LEN(head)+2
foot_len = LEN(foot)+2

IF head_len > view_len THEN view_len = head_len
IF foot_len > view_len THEN view_len = foot_len
OPEN "VOC" TO filevar ELSE
   FSMSG()
   STOP
END

SELECT filevar

/* initialize View window -- must be done before
   calling the HEADING or FOOTING statements */
@VIEW.MODE = 1
INIT.VIEW(view_len)
HEADING head
FOOTING foot
done  = 0

LOOP
   READNEXT @ID ELSE done = 1
  UNTIL done
   READ @RECORD FROM filevar,@ID THEN

/* Set column widths for the first two columns,
allowing one extra space for each column (allocated
above). */

   PRINT @RECORD<1> "L#11":" ":
   PRINT @RECORD<2> "L#26":" ":

/* No column follows the third, so don't add 1 to
column display length. */
      PRINT @RECORD<3> "L#15"
   END
REPEAT
STOP
```

INPUT.CHAR

INPUT.CHAR is the fundamental subroutine in Advanced Revelation for processing keyboard input. It checks all input characters for reserved functions, which trigger certain system or user-defined processes.

Optionally, INPUT.CHAR will also return the hexadecimal ASCII code for any input character.

INPUT.CHAR also controls the timing of background indexing, if that feature is enabled. You may substitute your own routines for the system-supplied background index processing, using the INPUT.CHAR code and command option.

```
INPUT.CHAR([key])
```

Using INPUT.CHAR

When invoked, INPUT.CHAR looks at the next key pressed at the keyboard. INPUT.CHAR processes all keystrokes except those defined as priority interrupts in @PRIORITY.INT (see EDIT.KEYS in the SYSINCLUDE file), and those keys that are subject to environment restrictions (for example, when the Command window is disabled). For all other characters, you may use INPUT.CHAR to return a one- or two-byte ASCII code. For example, [F4] will return an ASCII code of 00 62, but [F5], a priority interrupt, will invoke the next level of the Command window.

Background index processing is timed by the frequency of user input. INPUT.CHAR can be used to replace normal background indexing with a code and command process that uses your routines to handle background indexing (the default may be set by accessing Management-Environment-Index from the main menu). If fields 77 and 78 of @ENVIRON.SET are set, then when background indexing is to occur, the code and command process specified in those fields will occur instead of the system-defined background index processing.

key You may pass an argument for *key* to return the ASCII code of the next key pressed. Function keys return two bytes, where the first byte is 0. Whether or not a value is supplied for *key*, the values of pressed keys are passed into the system, after processing by INPUT.CHAR.

Once invoked, INPUT.CHAR does not return until a key is pressed.

Values returned

If an argument is provided for *key*, it will return the ASCII value of the next key pressed.

Last input character

The last character returned by INPUT.CHAR is available in the system variable @PROG.CHAR. This does not include any characters (such as the keystrokes for [F5]) that are actually proceeed by INPUT.CHAR.

The following code segment demonstrates how access to @PROG.CHAR can tell you whether the user pressed an escape key.

```
...
$INSERT SYSINCLUDE, EDIT.KEYS
...
@ANS = POP.UP(col, row, file, display, format, mode, ⇨
   select, title, attributes, help, coordinates, type)
IF @PROG.CHAR = @INT.CONST<QUIT$> THEN
   MSG("You pressed Escape...")
END ELSE
   TEXT     = "You didn't press escape,"
   TEXT<-1> = "and your return values are|%1%"
   MSG(TEXT, "", "", @ANS)
END
```

Correct use of INPUT.CHAR

The following code constructs a loop that asks for a keystroke, then reports the value of that key. This code also replaces standard system background indexing with the user-supplied subroutine DUMMY.

```
DECLARE FUNCTION ESC.TO.EXIT
DECLARE SUBROUTINE INPUT.CHAR, MSG

$INSERT SYSINCLUDE, ENVIRON.CONSTANTS
$INSERT SYSINCLUDE, EDIT.KEYS

EQUATE esc$ TO @INT.CONST<QUIT$>
done = 0
character = ""
image = ""
TEXT = "Please enter a character. Type [ESC] to quit:"
save_index_code    = @ENVIRON.SET<E.INDEX.CODE>
save_index_command   = @ENVIRON.SET<E.INDEX.CMD>
@ENVIRON.SET<E.INDEX.CODE> = "S"
@ENVIRON.SET<E.INDEX.CMD> = "DUMMY"
* user-supplied background index processing subroutine
LOOP
   MSG(TEXT,"UB",image,"")
   INPUT.CHAR(character)
   IF character = esc$ THEN done = 1
UNTIL done
   IF LEN(character) 1 THEN
      char1 = SEQ(character[1,1])
      char2 = SEQ(character[2,1])
      character = char1:" ":char2
   END ELSE
      character = SEQ(character)
   END
```

(continued)

```
TEXT = "ASCII code for the keystroke pressed is: %1%"
MSG(message,"","",character)
MSG("","DB",image,"")
REPEAT

@ENVIRON.SET<E.INDEX.CODE> = save_index_code
@ENVIRON.SET<E.INDEX.CMD> = save_index_command
STOP
```

MSG

MSG is the fundamental subroutine for message posting in the Advanced Revelation environment. MSG can also be used to accept input from the user.

```
MSG(message, map, response, values)
```

Using MSG

The great power and utility of MSG as a tool for displaying messages derive from the underlying message record structure that is a default when MSG is called. You may alter any part of that structure, but because it exists, you can display a message as simply as:

```
MSG("Hello, World!")
```

Learning to use MSG to its fullest capacity requires learning how to take advantage of each aspect of the record structure. That record structure can be selectively modified by the *map* parameter, described below.

message

The value of *message* may be simply literal text, as in the "Hello, World!" example just cited. The value of *message* may also be a record key to a message in one of the system message files (MESSAGES or SYSMESSAGES), or a record key in a user-supplied message file. MSG looks first in @MESSAGES to see whether the requested message is already in memory. Failing that, it looks in MESSAGES (a system-supplied message file intended for user messages), and then searches SYSMESSAGES. If the value of *message* is not a record key in one of these locations, the value of *message* is presumed to be a literal. The process of determining whether the value of *message* is in fact a literal can be circumvented by manipulating the value of *map*<13>, described below.

If *message* is literal text, you can break the message into multiple lines by including a "|" or a text mark (@TM) where you want the line to break. Value marks (@VM) and field marks (@FM) also break a display line.

When constructing text for messages, you may make use of any of several replaceable parameters, each beginning and ending with a "%" (percent sign). When the text

prints to screen, each parameter is expanded as specified. For example, if you include the replaceable parameter %T% in your message text, when the message is displayed, the "%T%" symbol will be replaced by the current time.

Several replaceable parameters derive their values at expansion time from the contents of *values*, described below.

Parameter	Meaning
%n%	Replaced by the *n*th element in *values*. [%0% is the entire value of *message*]
%D%	Replaced by the current date.
%G%	Replaced by all values in *values*.
%T%	Replaced by the current time.
%B%	Rings the bell.
%Cn%	Replaced by ASCII character *n*.

If MSG determines that *message* is literal text, *message* is transferred to *map*<mscript>, described below.

map The value of *map* is a message record, each field of which is @FM-delimited.

Field	Mnemonic	Function	Default
<1>	mtype	Message type	A
<2>	mbuffer	Buffer in @MESSAGES?	null
<3>	mcol	Left column	null
<4>	mrow	Top row	null
<5>	mattr	Video attribute	@EW<6>
<6>	mjust	Justification	C
<7>	mwide	Width of message	calculated
<8>	mborder	Type of border	@EW<1>
<9>	reserved		
<10>	reserved		
<11>	mscript	Message text	null
<12>	mchar.cnt	# of chars for response	to 64K
<13>	mmsgloc	Message record location	null
<14>	mmsgfile	File for message record	null
<15>	val_pattern	Response validation pattern	null
<16>	resp_code	Response code	null
<17>	resp_command	Response command	null

The message record structure is normally stored in a file (MESSAGES, SYSMESSAGES, etc.). The record structure is used when a message is called out of that file, and you can override all or part of that structure by supplying the affected fields in *map*.

For example, message 401 is stored as an "A" type message (field 1). This means that when displayed, the user must press a key to remove it. But if the only argument for *map* is a substitution of "T2" for the original "A":

```
MSG("401", "T2")
```

the only difference will be that the message is now displayed for only two seconds.

Characteristics and specifications for each field are presented below.

Field 1: Mtype

Message type. The following table shows available message type designations and their characteristics.

Specifier	Description
A	Display a bordered message. You are prompted to press a key to acknowledge and remove the message.
AI	The same as "A", but avoids the use of INPUT.CHAR to process the user response. This prevents such consequences of INPUT.CHAR as background index processing and access to @PRIORITY.INT keys.
AIX	The same as "AX", but avoids the use of INPUT.CHAR to process the user response. This prevents such consequences of INPUT.CHAR as background index processing and access to @PRIORITY.INT keys.
AX	Display a bordered message. You are *not* prompted to press a key to acknowledge and remove the message, but the system waits until you do. In the message, you should include instructions for the user on how to continue.
C	Substitute a call to CATALYST for this MSG call. The code and command arguments will be the value of *message*. Separate *code* from *command* with a value mark (@VM) or "|".

(continued)

Specifier	Description
D, DB	Removes a message that was displayed with "U" or "UB". The display image that is to be removed must be the value of *response* during this call to MSG.
H	*Message* is displayed without a border and without regard to justification, beginning at the location specified in <mcol> and <mrow> and defaulting to the cursor position at the time MSG is called. User input is not requested and the message remains on the screen until overwritten by another process.
HR	This option differs from "H" in waiting for a user response and allowing editing of the response. User response can be made subject to validation or conversion. See Field 15 below.
HRC	This option is the same as "HR", but converts lowercase response to uppercase.
L	This option is identical to "H", but inserts a blank line after the last line of *message*.
LR	This option differs from "L" in waiting for a user response and allowing editing of the response. User response can be made subject to validation or conversion. See Field 15 below.
LRC	This option is the same as "LR", but converts lowercase response to uppercase.
N	No message is displayed. In an application, this value could be changed dynamically. You may also edit SYSMESSAGES (system messages supplied by Advanced Revelation) to change *map*<mtype> to "N" and prevent the display of particular system messages.
R	*Message* is displayed along with a separate edit line for a response. User response can be made subject to validation or conversion. See Field 15 below.
RC	This option differs from "R" in converting any lowercase response to uppercase.

(continued)

M MULTI-PAGE SUPPORTS MWIDE PARAMETER
UP TO 20 LINES PER SCREEN DUE TO BORDER,
STATUS LINE AND BLANK LINES.

Specifier	Description
RE, RCE	These options are the same as "R" and "RC", but allow return *in response* of the ASCII code for the key currently defined as the ESC key. The calling routine can then determine that the user has purposely typed the [Esc] key (as, for example, a signal to cancel a process).
RI	This option differs from "R" in accepting input by means of the R/BASIC INPUT statement rather than INPUT.CHAR. This means that the input processing normally part of INPUT.CHAR is not in effect (background indexing, etc.).
S	*Message* is displayed in accordance with all specifications in the *map* record, but remains on the screen.
T*n*	*Message* is displayed for *n* seconds.
U	With this option, MSG first saves the current display image in *response*, then displays *message*. The contents of *message* remain displayed until the value of *response* is redisplayed using the "D" or "DB" option, which restores the screen to its condition before invoking the "U" option.
UB	This option differs from "U" in also displaying a border, as specified in *map*<mborder>.

Field 2: Mbuffer

The value of this field determines whether the message is to be stored in the message buffer (@MESSAGES). If so, subsequent access to this message is had through @MESSAGES.

Value	Function
null	Message is not buffered.
B	Message is buffered after display.

To force specific messages to be loaded into memory beginning at login, include their keys in the IN.MEMORY record of the SYSMESSAGES file. This record is read during the login process.

Field 3: Mcol

Use this field to specify the left column of the displayed message. The default (null) presumes horizontal centering.

Field 4: Mrow

Use the value of *map*<mrow> to specify the top row of the displayed message. This row is the border, if one is specified. The default is vertical centering.

Field 5: Mattr

Use *map*<mattr> to pass an escape sequence specifying a video attribute for the displayed text. Alternatively, you may use existing attributes as contained in one of the system video variables (@PW, @EW, @AW, etc.). To specify a system video variable, include the name of the variable as a literal ("@PW", "@EW", etc.).

Normal uses for system video variables are as follow:

Variable designator	Normal function
@EW	MSG messages
@HW	Help windows
@MW	Menu windows
@PW	Popup windows
@XW	Edit and Zoom windows

For example, to replace the normal video attributes of MSG with those of popup windows, use the following code:

```
message   = "B102"
map       = ""
map<5>    = "@PW"
MSG( message, map )
```

Blinking attributes You can now cause these "borrowed" video attributes to blink. To achieve this effect, add "at" symbols (@) before the "@*x*W" literal value:

Add	to achieve
@	a blinking border
@@	blinking text
@@@	both blinking border and blinking text

For example, to create a flashing border in your posted message, add one "at" symbol before the system video variable:

```
map       = ""
map<5>    = "@@PW"     *; flashing border
```

To create flashing text in your posted message, add two "at" symbols before the system video variable:

```
map       = ""
map<5>    = "@@@AW"       *; flashing text
```

To create both flashing text and also flashing borders, add three "at" symbols before the system video variable:

```
map       = ""
map<5>    = "@@@@AW"   *; flashing text and border
```

To just add blinking to MSG default video attributes, you must specify the "@EW" literal value.

Field 6: Mjust

Specify the text justification of the displayed message by passing one of the following literal values:

Literal value	Function
L	Left justified.
R	Right justified.
C	Centered.

Field 7: Mwide

You may use *map*<mwide> to specify either the width or the width and height of displayed text. By specifying height (the second, @TM-delimited, element in this field), you can effectively page your displayed text.

The first value in *map*<mwide> governs the width of the displayed text. To specify the number of rows (lines) displayed at one time, include an @TM and a value for height. If you specify height (the number of rows), you cannot use a null value for text width. If there is an @TM in this field, you must include an integer for width before it.

Field 8: MBorder

MSG recognizes values 1-5 for this field. To specify a title for a displayed message, append @TM and the title to the border type. If you are including a title, the value of the border type can be null.

Field 9: Reserved

Field 10: Reserved

Field 11: Mscript

Use *map*<mscript> to specify the text to be displayed. You may use "|" or @TM to force a line break. Otherwise, line length will be determined by the value of *map*<mwide>.

If the value of *map*<mtype> is "C", the value of *map*<mscript> will be interpreted as a code and command sequence to be run by CATALYST. Separate *code* from *command* with either a "|" or a value mark (@VM).

Field 12: Mchar.cnt

Use *map*<mchar.cnt> to specify the number of characters allowed for a response (defaults to 64K). When the specified limit is reached, a carriage return is executed.

Field 13: Mmsgloc

Use *map*<mmsgloc> to specify how *message* is to be interpreted.

- If null (the default), MSG assumes first that *message* is a record key and looks for it first in MESSAGES (intended for developer-supplied messages), then in SYSMESSAGES (where Advanced Revelation stores its system messages). Failing to find the record in these two sources, MSG then presumes the value of *message* to be literal text.
- If *map*<mmsgloc> is 1, MSG skips the record key search and assumes that *message* is literal text.
- If *map*<mmsgloc> is 2, MSG consults a particular file. When the value of *map*<mmsgloc> is 2, the name of the file to be searched must be in *map*<mmsgfile>.

Mmsgloc value	Function
null	Search system files.
1	Interpret as literal.
2	Search user-supplied file.

Field 14: Mmsgfile

If the value of *map*<mmsgloc> is 2, MSG will search the file specified in *map*<mmsgfile> for the record the key of which is the value of *message*.

Field 15: Val_pattern

Use *val_pattern* to specify a validation function or custom conversion or validation subroutine that will handle responses to MSG message types that provide a user response in *response* (HR, L, and R). One use for this feature is to handle foreign language responses to existing messages that require an English-language response (for example, "Yes/No"). A custom routine could convert the foreign language response into the system- required "yes" or "no". Validation routines could handle conversion of dates, numerics, and currencies into Advanced Revelation internal formats.

Case conversion occurs before the user-supplied conversion routine is called. If the validation or conversion routine returns invalid, the code and command processing specified in fields 16 and 17 is automatically invoked, if it exists.

For more information on how to specify conversion functions and code conversion subroutines, see "User-Defined Conversions" in this chapter.

Field 16: Resp_code

Use *resp_code* to provide a code operand, which together with the command specification in field 17, will be automatically invoked when user response is invalid. Specify "NONE" if you do not want any error messages, including system error messages, to be invoked upon invalid user response. If this field is null, system error responses, if any, are invoked upon invalid user response. These system responses typically allow editing of the user response.

Field 17: Resp_command

Use *resp_command* to supply the command specification for the code operand provided in *resp_code*.

response The variable assigned as the argument for *response* is used by MSG to return values. You can use the variable initialization step to provide a suggested value for a user response.

Response has two major functions: to hold user response when appropriate, and to hold the value of a screen display before MSG alters that display. If you save the screen display in *response*, you can restore the original display with a "D" argument in *map*<mtype>, effectively erasing the text MSG has just displayed.

Map<mtype> value	Function of response
U, UB	Returns display image.
HR, HRC, R, RC, RE, RI	Returns a user response.
D, DB	Passes a display image.

values Use *values* to pass values that will replace any replaceable parameters specified in *map*<mscript> or in *message*. Separate multiple values (for multiple replaceable parameters) with "I", @VM, or @FM. The position of elements in *values* must reflect the order of their corresponding replaceable parameters in *message* or *map*<mscript>.

Values returned

MSG returns in *response* any values it provides to the calling routine. These include a display image and a user response.

Redirecting system messages

You can use MSG, in combination with CATALYST, to intercept and customize system messages, making use of the internal values that are passed to system messages.

If you use the C operator of CATALYST, you can make use of the following process sequence followed by MSG:

1. The contents of @ANS are temporarily saved
2. System-supplied values are passed to @ANS
3. A CATALYST call can invoke a subroutine, which generates a new value for @ANS
4. The new value of @ANS is transferred to the MSG RESPONSE argument
5. The stored contents of @ANS are returned to @ANS

There are two important consequences of this sequence:

- Access to the MSG VALUES argument, when the system has supplied the values
- Ability to pass a previously defined value in the MSG RESPONSE argument

How it works

The system passes values to system messages that contain replaceable parameters. For example, system message W156 reads:

```
"%1%" is an unrecognized word.  Please correct the word
by retyping it or pressing [F4] to edit it.  Press
[Esc] to restart.
```

When the system calls MSG to display this message, it passes a value for %1% in its *values* parameter (the 4th MSG argument). In this example, the system passes the misspelled TCL command.

MSG now places the contents of the *values* argument in @ANS. Therefore, as MSG is executing, you can pass these system-supplied values to a custom subroutine that either displays these values in its own way, or that uses these values differently than intended.

To redirect system messages, have MSG execute a subroutine instead of displaying a message. Do this by making the value of the first field in the *map* argument (*mtype*) "C" ("call CATALYST"), and pass as the value of *message* whatever CATALYST code and command you want. For example, to have MSG run a CATALYST code and command process when a particular system message is invoked, edit the record in SYS.MESSAGES:

1. change the "R" value of the first field to "C"
2. change the value of field 11 to a @VM-delimited code and command sequence

You can use the call to CATALYST either to display your version of the message, or to take some other action. The values the system would have used for the replaceable parameters are available in @ANS. In addition, you can provide values for the RESPONSE argument (the 3rd MSG parameter).

For example, if you are going to replace the W156 system message mentioned above, you might pass the system-supplied values to a subroutine called SUB_SYSTEM. In that case, the value of the 11th field of the W156 record in SYS.MESSAGES would be

S^{2}SUB_SYSTEM

where "S" is separated from "SUB_SYSTEM" by a value mark.

In this example, SUB_SYSTEM could examine the system-supplied values, and substitute a TCL command for the attempted command text.

Another example would modify the message that displays when you give the command to delete a file (message W512). This message passes the name of the file you are attempting to delete. If you have MSG invoke CATALYST with a subroutine call, then that subroutine could write the user name, time and date, and file name to a security or record-keeping file.

Alternative message files

Normally, MSG searches two files to match the value of *message* to a record key in a system message file: MESSAGES and SYSMESSAGES. You can circumvent this search process by specifying the name of a file and the record from that file to display.

One use for this would be to keep application-specific messages in a file separate from system messages, especially if you have more than one application-specific message file.

To use this feature, make the value of *message* the name of the file, where that name is surrounded by "at" symbols, and the record key follows immediately after the second "at" symbol. For example,

```
MSG( "@file_name@record_key" )
```

would display the message associated with the *record_key* record in the *file_name* file.

Supressing message numbers

Record keys normally display as a title in the border of the message. For example, message W156 has "W156" as its title. You can now prevent display of the record key value.

Add the value "N" to the message type (*mtype*, the first field of the *map* argument). For example, if your message type is currently "RC", change it to "RCN". This works for any message type.

Correct use of MSG

The following program demonstrates various functions of MSG.

```
DECLARE SUBROUTINE MSG, DELAY

EQUATE mtype$ TO 1
EQUATE mbuffer$ TO 2
EQUATE mcol$ TO 3
EQUATE mrow$ TO 4
EQUATE mattr$ TO 5
EQUATE mjust$ TO 6
EQUATE mwide$ TO 7
EQUATE mborder$ TO 8
EQUATE mscript$ TO 11
EQUATE mchar.cnt$ TO 12
EQUATE mmsgloc$ TO 13
EQUATE mmsgfile$ TO 14

* 1st example: mtype = A (the default)
*  Show a message, ask for a response to continue

GOSUB clear_values
message = "W964"  ;* record key in SYSMESSAGES
map<mtype$> = "A"
values = "SAMPLE_FILE":@FM:"SAMPLE_VOLUME"
MSG(message, map, response, values)
```

(continued)

```
* 2nd example: mtype = AI

GOSUB clear_values
message = "W936"
map<mtype$> = "AI"
MSG(message, map, response, values)

/* 3rd example: mtype = C  * Invoke CATALYST with value
of MESSAGE */

GOSUB clear_values
map<mtype$> = "C"
message   = "E|LIST SAMPLE_CUSTOMERS"
MSG(message, map, response, values)

/* 4th example: mtype = H  * Post a message, then
return to processing */

GOSUB clear_values
message = "Here is some text"
map<mtype$> = "H"
map<mcol$> = 5
map<mrow$> = 20
MSG(message, map, response, values)

/* 5th example: mtype = HR  Post a message, then wait
for a response */

GOSUB clear_values
DELAY(5) ;* give time to notice difference
message = "Here is some more text"
map<mtype$> = "HR"
MSG(message, map, response, values)

/*6th example: mtype = R  Post a formatted message,
asking for response.*/

GOSUB clear_values
message = "104"
map<mtype$> = "R"
MSG(message, map, response, values)

/* 7th example: mtype = Tn  Post a timed message (5
seconds).*/

GOSUB clear_values
message = "This message will be timed...."
map<mtype$> = "T5"
map<mattr$> = "@AW"
MSG(message, map, response, values)
```

(continued)

```
/* 8th example:mtype = U  Post a formatted message,
saving previous screen in RESPONSE. */

GOSUB clear_values
message = "This message must be removed with
map<mtype$>='D'"
map<mtype$> = "UB"
map<mborder$> = 2:@TM:"Message Title"
MSG(message, map, response, values)

/* 9th example: mtype = D  Redisplay the screen from
before the 'U' MSG call. */

DELAY(5)
message = ""
map<mtype$> = "D"
MSG(message, map, response, values)
STOP

**********************
* Internal Subroutines
**********************
clear_values:
message = ""
map = ""
map<mtype$> = ""
map<mbuffer$> = ""
map<mcol$> = ""
map<mrow$> = ""
map<mattr$> = ""
map<mjust$> = ""
map<mwide$> = ""
map<mborder$> = ""
map<mscript$> = ""
map<mchar.cnt$> = ""
map<mmsgloc$> = ""
map<mmsgfile$> = ""
response = ""
values = ""
RETURN
```

POP.UP

POP.UP is a function that creates and displays a popup, from which it returns user-selections.

```
return_selections = POP.UP(col, row, file, display, ⇨
    format, mode, select, title, attributes, help, ⇨
    coordinates, type)
```

Using POP.UP

Designing popups can be a complex process, because POP.UP interprets arguments for parameters differently, depending upon arguments for certain basic parameters, such as *mode*. For example, an argument for *display* is interpreted differently, depending upon whether the popup is to display as a table of records, as a matrix of fields and values from one record, or as a matrix of values and subvalues from a single field in a record. For detailed information on the structure of popups, see "Creating popups" in the *User's Guide*.

col Use an argument for *col* to specify the starting column for the popup. If null, the default is 0.

row Use an argument for *row* to specify the top row for the popup. If null, the default is 0.

file Use an argument for *file* to provide the source of data to display within the popup, when that data is not supplied by a select list or by values in *display*.

There are three possible arguments for *file:*

Argument	Function
@filename	POP.UP will attempt to open the specified file.
file_variable	POP.UP will use a previously opened file.
null	POP.UP assumes literal data in *display*.

display Use an argument for *display* to determine how the data from *file* is to be interpreted and displayed, or, if *file* is null, to provide the data itself.

The format of arguments for *display* depends upon the argument for *mode*.

If *mode* is T (Table), *display* can contain:

Mode T display argument	Comment and example
Dynamic array of record keys	When *file* is not null; record keys must be @FM-delimited. `"ENTRY":@FM:"CONTACTS":@FM:"REPORTS"`
Null	Current select list is used; if a select list is not available and *file* has QUICKDEX attached, the QUICKDEX order of records applies.
Literal data	*Format* is null and *display* should be a dynamic array of @RM-delimited records, which format as rows, and @FM-delimited fields, which format as columns.

If *mode* is R (Record), *display* can contain:

Mode R display argument	Comment and example
Single record key	When *file* is not null, the fields of the specified record display as the first data column of the popup. `"INV_TOTAL"`

Note When *mode* is R and *display* contains more than one value, *file* is presumed null and *display* is treated as having literal data.

If *mode* is F (Field), *display* can contain:

Mode F display argument	Comment and example
Dynamic array of literal data	When *file* is null, @FM-delimited fields display as rows and @VM-delimited values as columns. `"Ralph":@VM:"inv":@FM:"Mary":@VM:"payroll"`
Single record key and field	When *mode* is F, the specified field will number or name make up the first data column of the popup. `"INV_TOTAL":@FM:6`
Dynamic array of literal data	When *file* is null, @VM-delimited fields display as rows and @SVM-delimited subvalues as columns.

```
"Sam":@SVM:"orders":@VM:"Jan":@SVM:"stock"
```

format Format defines the appearance of each column in the popup. The *format* parameter is structured as follows:

```
field:width:just:oconv:heading
```

Each five-element string governs the display of one column. Each of the five positions must be separated one from the other with ":" (colon), where the colons are necessary even when the subargument is not passed. The *format* subarguments must be passed as a literal string. Define format specifications for a multi-column popup by linking multiple specifications with "\" (backslash). For example,

```
format = "1:12:L::Action\2:25:L::Description"
```

would govern the display of two data columns, where the *oconv* argument is not specified for either column.

field *Field* contains the field number or field name from *file* or *display*, the contents of which are to be displayed in the specified column. If *mode* is T, a 0 in *field* indicates the key field.

width The value of *width* governs the display width of a data column.

just The value of *just* governs the justification of displayed data. C (centered), L (left),and R (right) are possible values. When *just* is null, the default is "L".

oconv The value of this argument, if specified, must be a valid OCONV conversion pattern (see R/BASIC OCONV). When null, there is no default conversion.

heading This is the heading of the data column. When null, there is no default heading.

mode *Mode* determines the scale of the popup display: whether it is a table, a record, or a field.

Mode	Meaning
T	Table (exhausts the active select list)
TS	Table (first saves active select list, restoring afterward)
R	Record
F	Field

select The value of *select* determines how the user is allowed to select the data in the popup. If the value of *type* is R (read only), an argument for *select* has no effect.

null or 0 *Single selection*: user presses [Enter] to return a popup selection.

1 *Multiple selection*: user presses [Enter] to select, then [F9] to return the selections. Selections are returned as position numbers in the order selected.

G *Groupings*: the user determines what data columns are to be grouped together by pressing [Enter] at each selection. See "Grouping and Reordering Prompts" in the "Adding Advanced Features to Windows with Paint" chapter of *Developing an Application* for more information on this type of *select*.

The format of the G subargument of the *select* variable is:

```
G:@FM:data_column:@FM:starting_group
```

here G is the type of select, *data_column* is the field in which the group number is to reside upon return (replacing whatever was there before), and *starting_group* is the initial group number presented to the user (default is 1).

The value of *data_column* determines which field will contain the assigned group number upon return. You must then check the value of this field to determine in which group the user has placed this data item. For example, if you specify that *starting_group* will be "3", the popup will assign a "3" to each selected row. When the popup returns its values, that "3" will be in the popup field specified by the value of *data_column* (default is 1). Because the popup returns all data from the popup display, regardless of whether it was selected, only by checking the correct field (value of *data_column*) can you determine whether a row was grouped and, if so, in which group it was placed.

O *Ordered multiple selection*: the user determines the order in which the selections are returned by pressing [Enter] in the new order. An additional column, labeled "Order", is added to the popup display. As the user presses [Enter], a number representing the position in the desired order appears in this column. Only selected data is returned by the popup (the field corresponding to the value of *coordinates*), and it is returned in the order selected.

S Sort data rows: the user determines the order of the popup display by pressing [Enter] at each row to change, then uses the arrow keys to move the label to the desired row, where another press of [Enter] causes the change in display order. A Sort Column is added to the popup display. No values are returned.

The S subargument has a four-field structure:

```
S:@FM:data_column:@FM:starting_row:@FM:ending_row
```

where S is the type of select, *data_column* is the column from which data will be displayed in the Sort Column, *starting_row* is the row at which to begin, and *ending_row* is the final row that can be sorted. For example,

when *data_column* is "1" and column one of the selected row has "Sam", "Sam" will appear in the Sort Column until that row has changed position.

T *Toggled selection*: the user selects popup rows by pressing [Enter], where [Enter] is a toggle, first selecting, then rejecting. Each selected row will contain either "yes" (the default tag) or a tag specified in field 4 or 5.

In this mode, only the "yes" or other tag is returned, but a null field is returned for each selection not selected when those empty selections are necessary to preserve the significant null fields among the returned values.

The T subargument has a five-field structure:

```
T: @FM: return_field: @FM: display_column: ⇨
   @FM: response_1 :@FM: response_2
```

The T argument for the *select* parameter returns data from column 1 and the response to each row, if any. The response will be in whatever field of the returned array is specified in *return_field* of the *select* argument (default is 2). That response will be displayed in the popup in whatever field is specified in *display_column* of the *select* argument.

Response_1 and *response_2* each allow you to specify a particular response when the user presses [Enter] at a given row. With the first press, the value of *response_2* appears in the column of the popup specified in *display_column*. When that row is toggled, the value of *response_1* takes its place. A row that has not had [Enter] pressed by it will not have its value, if any, changed in the column specified in *display_column*, but if both *response_1* and *response_2* specify values, once a response has been toggled, it cannot be removed.

title Use *title* to display a title in the top row of the popup. If you want to display a multi-line title, delimit text with either "|" or @FM.

Justification of the title defaults to "centered". You may override this by appending the literal value "L" or "R" to the title data (separating that value from the data with @SVM), but be aware that this change only affects the last line of a multiple-line title.

attributes The default value of *attributes* is the value of @PW.

help Use an argument for *help* to specify a record in a system help file that provides help for the popup when [F1] is pressed. HELP is searched first for the specified record, then the SYSHELP file. If found, POP.UP runs CATALYST with an HF code. Pass a null argument if you do not want help for [F1].

Optionally, if the value of *help* is comma-delimited, POP.UP knows that the value of *help* is a "code,command" command string for CATALYST.

For more information on using CATALYST, see that documentation earlier in this chapter.

Support for row-sensitive popup help

The *help* parameter of POP.UP can also be used to call row-sensitive help.

The command portion of the *help* code and command argument can include one of the following three literal values:

argument	returns
<@POS>	value in first column of current row
<@ROW>	current row number
<@n>	value of specified column in current row, where *n* is the number of the column to return

For example, to call a help record with the row-dependent key position from either the SYSHELP or HELP table, you would pass the following argument for *help*:

```
help = "HF,<@POS>"
```

coordinates Use an argument for *coordinates* to determine the column from which data will be returned by POP.UP. The value of *coordinates* is subordinate to that of *type* when there is a conflict.

There are two possible fields for the value of *coordinates*:

```
data_column:@FM:data_row
```

The value of *data_column* determines which column or columns of data will be returned by the popup. The default value is 1. Pass a value mark-delimited list of column numbers, to return values from multiple columns. Multiple return values are also value mark-delimited. If the popup is a multiple selection popup, each row is returned as a separate field (field mark-delimited).

The value of *data_row* determines at which row the cursor is positioned when the popup first displays. Default is 1.

data_row returns the number of the row where the cursor was positioned when the user exited the popup. You can use this position when calling POP.UP again, to initially position the cursor. This field of *coordinates* is used to keep continuous track of the cursor position.

data_row is fully functional in R mode popups. In T mode popups, an argument is valid only for single-screen popups or only for the first screen of a multi-screen popup.

type Use an argument for *type* to specify the kind of data that is returned for each user selection. *Type* specifications override any conflicting specifications in *select* or *coordinates*. When *type* is null, the data column and/or row are determined by the value of *coordinates*.

K, R, P, and null are the possible values for the first field. That value can be augmented by subvalues. Ten subvalues are accepted for this argument.

Type<1,1,*n*>, where *n* is one of the following subvalues:

Type field	Value and function
type<1,1,1>	Four arguments are valid:
K	Record key(s) of selected data are returned. Only valid in table mode.
R	Read only. No values are returned.
P	Selection row numbers are returned.
null	The data from a selected row is returned (an entire display row).
type<1,1,2>	If true, POP.UP will leave the popup displayed after the function of the popup has been achieved.
type<1,1,3>	Number of rows per screen.
type<1,1,4>	If true, the popup will return the image beneath the popup for later use by VIDEO.RW.
type<1,1,5>	Reserved.
type<1,1,6>	This may contain a list of rows to be selected by default; must contain @TM-delimited integers.
type<1,1,7>	Reserved.
type<1,1,8>	If this parameter contains a comma-delimited code and command, CATALYST executes that code and comand when the user presses [Ctrl-F2] from within the popup.
type<1,1,9>	Setting this to true causes the code and command stored at <1,1,8> to be executed automatically after display of popup, but before the user is allowed to enter anything.
type<1,1,10>	If this parameter contains a comma-delimited code and command, the code and command is executed when [F3] is pressed.

Adjusting popup display

The 14th value of the POP.UP*PARMS record in the SYSTEXT system table specifies the display width of the number column at left in a popup. The default value is 4, but you can change this to any value less than @CRTWIDE-10.

Popup data from subroutines

You can issue a code and command statement (for CATALYST processing) to provide data for popups. This can be especially helpful for server connections, where all rows

to be displayed in a popup can be returned in one process, rather than issuing a select statement for each row of the popup. The principle can be extended as well to allow you to programmatically provide rows of data for popups from non-server sources, such as a BTREE.EXTRACT selection from a Linear Hash file.

Implementing codes and commands

To invoke a code and command process from within POP.UP, use the *display* argument to pass the required data in the following format:

@code command@

Notice that a value for *code* is separated from the value for *command* by a space.

Other data available to a user process

Information in @ANS about the file and columns used is available to a user process:

@ANS field	Data available
1	Advanced Revelation file name
2	File handle for file in field 1
3-*n*	Multivalued column information

This information is valid for the file passed as the *file* argument. The column formatting information is taken from the value of *format*.

Popup behavior

Because T-mode popups use a select list, the process you call with your code and command must generate a select list.

If you create an R-mode popup, your routine must place data to be displayed in @ANS. See the description of how to use the *display* parameter in the documentation for the POP.UP system subroutine, for information on the structure of this data.

The user process is not available for F-mode popups.

Correct use of POP.UP

The following code demonstrates various features of the POP.UP function.

```
DECLARE SUBROUTINE MSG
DECLARE FUNCTION POP.UP

col    = 0
row    = 0
file   = ""
display  = "Edit file":@VM:"File to edit"
display<-1> = "Delete file":@VM:"File to delete"
format   = "1:12:L::Action\2:25:L::Description"
mode   = "R"
select   = ""
title = "Utilities|File operations"
attributes = ""
help   = ""
coordinates = 1
type   = ""

/* The following code demonstrates specification of a
two-column, single selection, unformatted popup that
returns the user selection in @ANS */

@ANS = POP.UP(col, row, file, display, format, mode,⇨
   select, title, attributes, help, coordinates, type)

MSG("The user selection is: |":@ANS)

/* The next code displays a two-column, multiple
selection, formatted popup, and specifies the type of
user-selected data to be returned in @ANS */

select = 1
mode   = "F" ;* field by values
type   = "P" ;* return numbers of selected rows

@ANS   = POP.UP(col, row, file, display, format, mode,⇨
   select, title, attributes, help, coordinates, type)

MSG("The user selection is: |":@ANS)

/* The following code displays a three-column group
selection, formatted popup, and specifies 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"
```

(continued)

```
@ANS = POP.UP(col, row, file, display, format, mode,⇨
   select, title, attributes, help, coordinates, type)

MSG("The user selection is: |":@ANS)

/* The next code displays a three-column, KEY
selection, table-formatted popup, for which the
SAMPLE_PRODUCTS file from the SAMPLE_APP volume
provides data. */

file = "@SAMPLE_PRODUCTS"
display = ""
title  = "Product List"
format = "0:5:L::Code\1:20:L::Description\10: ⇨
   6:C::Order"
mode = "T"
select = "1"
type = "K"

$INSERT SYSINCLUDE, WINDOW.POINTERS

* see what kind of video display we're using
MONITOR(status)
IF status = 7 THEN   ;* i.e., we are monochrome
   EQUATE esc$ to CHAR(27)
   attributes<cborder>  = esc$:"C7?"
      * rev. highlight
   attributes<ctitle>   = esc$:"C78"
      * rev. low-intensity
   attributes<cbar>  = esc$:"C01"   ;* underline
   attributes<cborder.type> = 1  ;* double lines
END ELSE ;* let's presume color
   attributes<cborder> = esc$:"C":\030E\
      * yellow on cyan
   attributes<ctitle> = esc$:"C":\0007\
      * white on black
   attributes<cbar>  = esc$:"C":\0607\
      * white on brown
   attributes<cborder.type>   = 1
END

@ANS = POP.UP(col, row, file, display, format, ⇨
   mode, select, title, attributes, help, ⇨
   coordinates, type)

MSG("The user selection is: |":@ANS)
```

(continued)

```
/* The following code displays a three-column, SORT
selection, formatted popup, where video attributes are
specified. The user may change the order of the display
rows. No values are returned. */

file = "@SAMPLE_CUSTOMERS"
format_1 = "0:5:L::ID"
format_2 = "\1:20:L::Company name"
format_3 = "\PHONE_NO:15:L:L(###) ###-####:Phone"
format = format_1:format_2:format_3
mode = "T"
select  = "S":@FM:1:@FM:1:@FM:""
type = ""

@ANS = POP.UP(col, row, file, display, format, ⇨
   mode, select, title, attributes, help, ⇨
   coordinates, type)
```

PROGRESS

Provides a means to update the status line with information about the progress of a process.

```
PROGRESS( ActionCode, ActionMode, ActionData )
```

Using PROGRESS

PROGRESS requires three separate calls:

1. initialize
2. update
3. clean up

For the initial call, you store the contents of the status line before changing it.

For each update, you provide data to PROGRESS that allows it to either update a graphic image on the status line, or to post status text in a message box.

Upon conclusion of your routine, call PROGRESS with the clean up argument, which restores the status line to its condition before your routine began execution.

ActionCode Determines what the routine does. Pass one of the following codes:

0	initialize
1	update the status line
2	update a message box, not the status line
3	clean up

ActionMode Function depends on value of *ActionCode*. Pass one of the following values:

0	optional: name of calling routine (for purposes of customization); does not display
1	*progress_info;* this is a field mark-delimited array, where the first field is the current position in the processing. The second field is the quantity you assign as the total amount of processing to be performed. For example, if there are 900 rows to be processed, and the current position is 27, then the value of *progress_info* would be: 27:@FM:900.
2	text to be displayed, when the status line is not being used
3	null

ActionData Function depends on the value of *ActionCode*. This parameter is used to both pass values, and also to return values, depending on the value of *ActionCode*:

0	pass a variable to store the contents of the current status bar
1	any text to display in addition to the graphic "gas gauge" indicator; use %P% in this text to show where the percentage done numeric figure should appear within the text If you do not use %P% anywhere in this argument, only the graphic gas gauge will appear (no numeric percentage figure).
2	text to display; use %P% as above
3	data stored from the initialize call

Note The %P% percentage token is only available for status line updates. You cannot use it for a message box display (when the status line is not active).

The mutual dependencies for the various values of ActionCode can be summarized in the following table:

ActionCode	Function	ActionMode	ActionData
0	initialize	name of calling routine (optional)	variable to store current status line
1	update (status line)	*progress_info*	text in addition to graphic symbol
2	update (text in message box)	display text	display text (optional)
3	clean up	null	restore status line from variable

Example The following code segment demonstrates the use of Progress:

```
EQU init$ TO      0
EQU update$ TO    1
EQU cleanup$ TO   3

EQU max$ TO       100

DECLARE SUBROUTINE PROGRESS

PROGRESS( init$, "MYPROGRAM", hold )

FOR x = 1 TO max$
   /* ... do processing here */
   PROGRESS( update$, x: @FM: max$, "%P% Done" )
NEXT x

PROGRESS( cleanup$, "", hold )

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)
```

Using REDUCE

Use REDUCE to programmatically provide reduction criteria to a select list. You pass a specification statement in *reduce_script* that is similar to its R/LIST counterpart.

The R/BASIC extended select command (SELECT...BY) does not allow passing selection criteria, as you can with the TCL SELECT statement. REDUCE provides that capability. REDUCE used together with SELECT...BY gives you the equivalent in R/BASIC of the TCL SELECT process.

REDUCE uses the dictionary associated with a file to carry out its function, but you may optionally provide the name of another dictionary file in its place. See *file_name* below.

reduce_script

Use *reduce_script* to pass 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
Between	BETWEEN
Equal to	EQ
Less than	LT
Greater than	GT
Less than or equal to	LE
Greater than or equal to	GE
Not	NOT
Starting with	]
Ending with	[
Containing	[]
Matches	MATCH
Range	FROM TO
Sunday	SUN
Monday	MON
Tuesday	TUE

(continued)

Operator	Primitive
Wednesday	WED
Thursday	THU
Friday	FRI
Saturday	SAT
Today	TODAY
Yesterday	YST
Tomorrow	TOM
Next	NEXT
Last	LAST

Example assignments for *reduce_script*:

```
reduce_script = "WITH {DATE} NOT EQ '9-23-59'"
reduce_script = "WITH {CITY} EQ 'BOSTON' 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)
reduce_script := ' OR WITH {INVOICE_AMT} GE "1000"'
reduce_script = 'WITH {PHONE} MATCH "3N-4N"'
```

sort_list Pass the sort criteria in *sort_list*. 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 appropriate (when there are more than 100 records).

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 Use *mode* to pass 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 Use *file_name* to pass a file name and, optionally, an alternate dictionary to be used for that file. Separate the alternate dictionary name from the file name with a field mark.

Note Although this second field is the name of a dictionary file, do not place "DICT" before the file name.

cursorvar Use *cursorvar* to pass a cursor already initialized (MODES 0 and 2) or a cursor to be initialized (MODE 1). An explicit cursor number may also be passed. For more information on cursors, see "Cursors" in the "Programming with R/BASIC" chapter.

flag *Flag* returns true if the subroutine was successful and false if unsuccessful.

Correct use of REDUCE

The following code provides reduction criteria to a selection of keys from SAMPLE_CUSTOMERS using SELECT...BY. Two passes are made on SAMPLE_CUSTOMERS: in the first, the next available cursor is used to select keys by two values of the STATE field, sorted by those values. In the second, the same cursor is used (effectively erasing the previous contents) to select again by STATE, but sorting on the values of ZIP_POSTAL.

```
DECLARE SUBROUTINE REDUCE, MSG, FSMSG
EQUATE true$ TO 1
EQUATE false$ TO 0
EQUATE mjust$ TO 6
EQUATE state$ TO rec<6> ;* for SAMPLE_CUSTOMERS
EQUATE zip$ TO rec<7>   ;* for SAMPLE_CUSTOMERS
EQUATE new_exist$ TO 0  ;* REDUCE mode 0
EQUATE next_cur$ TO 1
EQUATE add_exist$ TO 2

file_name = "SAMPLE_CUSTOMERS"
flag  = ""  ;* REDUCE FLAG variable
done  = false$ ;* READNEXT terminator
cursorvar = ""
```

(continued)

```
* ensure that all cursors are available

FOR X = 1 TO 8
   CLEARSELECT X
NEXT X

* "%" means display status bar if appropriate
sort_list   = "%STATE"
reduce_script = "WITH {STATE} EQ 'NY' OR WITH ⇨
   {STATE} EQ 'CA'"

mode = next_cur$
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
OPEN file_name TO file_var ELSE FSMSG(); STOP

done  = false$
message  = "REDUCE selects STATE = NY and CA"
message<-1> = ""
message<-1> = "Key   State"
LOOP
   READNEXT key USING cursorvar BY AT ELSE done = true$
  UNTIL done = true$
   READ rec FROM file_var, key ELSE FSMSG(); STOP
message<-1> = key:"    ":state$
REPEAT

/* Resetting mode indicates that these are additional
criteria for an existing cursor. */

mode  = new_exist$
sort_list   = "%ZIP_POSTAL"
reduce_script = 'WITH {STATE} EQ "CA"'
REDUCE(reduce_script, sort_list, mode, file_name, ⇨
   cursorvar, flag)

IF flag ELSE
   MSG("Failure in REDUCE")
   STOP
END
```

(continued)

```
SELECT file_name BY sort_list USING cursorvar ELSE
   FSMSG()
   STOP
END

done  = false$
message<-1> = ""
message<-1> = "Now, selecting 'CA', sorting by the ⇨
   ZIP field:|"
message<-1> = "Key":"  ":"State":"  ":"Zip"

LOOP
   READNEXT key USING cursorvar BY AT ELSE done = true$
  UNTIL done = true$
   READ rec FROM file_var, key ELSE FSMSG(); STOP
   message<-1> = FMT(key, "L#9"):FMT(state$, ⇨
      "L#7"):zip$
REPEAT

MSG(message)
STOP  ;* end of routine
```

SAVELIST_SUB

Save an active select list as a row in a table.

```
SAVELIST_SUB( TableName, RowName, Options)
```

TableName Table in which to store the saved list. If null, LISTS is used.

RowName Row identifier (key) by which to store the saved list. Required.

Options Pass "A" to leave the current list active after saving.

Correct use of SAVELIST_SUB

```
SAVELIST_SUB( "", "QUERY101", "A" )
```

Saves the current active list as the QUERY101 row of the LISTS table, and leaves that list as the current active list.

SCRIBE

SCRIBE is Advanced Revelation's editor.

```
SCRIBE( inval, protect, exit_key, except_key, col, row, ⇨
    displen, color, flags, change, lines, exit, inpat, ⇨
    outpat, just, delim, mask, linelen, state )
```

Using SCRIBE

SCRIBE 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 through cursor control, multi-line entry, data validation and masking, and compatibility with Advanced Revelation-reserved keystrokes (such as [F5] for the Command window). 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 through 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.

inval Use *inval* to pass 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, *protect* prohibits the editing of *inval* data. A system message will be generated if editing is attempted. *Protect* defaults to false.

exit_key Use *exit_key* to pass the key or key sequence that will exit SCRIBE whenever editing is enabled. Multiple *exit.keys* must be separated by I or @FM. In addition, exiting will occur 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 Use *except_key* to pass the key or key sequence that will exit SCRIBE whenever editing is disabled. Multiple *except_keys* must be separated by I or @FM. In addition, exiting will occur if [↑] is pressed at the top of an edit window or [↓] at the bottom.

col Use *col* to pass an integer indicating the starting display column of the edit environment.

row Use *row* to pass an integer indicating the top display row of the edit environment.

displen Use *displen* to pass 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 Use *color* to pass 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 Use *flags* to pass a 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$>	Pre-process commands. This must be the ASCII code for the keystroke or keystroke sequence that calls the desired process. The function key [F1], for example, would be passed as CHAR(0):CHAR(59). The associated operation will be performed after *inval* is displayed and prior to the processing of the keystroke buffer (the value of @DATA).
<NOPAINT$> *(continued)*	Suppress display of text. If true, text is not initially displayed.

Field	Description
<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 *state*<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 with *state* specifications to redisplay text.
<NOSUBVALUES$>	Subvalue edit. If true, subvalue editing is disabled.
<SAVE.STATE$>	Save current state. If true, the current edit session parameters are automatically saved in *state*. SCRIBE will then update these parameters during the edit process.
<NOZOOM$>	If true, Zoom mode is disabled.
<NO.ORIG.VALUES$>	Suppress save of *inval* data. If false, and if *state*<SS.ORIG.VAL> is null, a copy of the original *inval* data is saved in *state* (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 *Change* returns true or false indicating whether edit changes occurred to *inval* data. False is returned if no changes were made, and true is returned otherwise.

lines Use *lines* to specify 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.
n	Full screen edit (where *n* is an integer indicating the number of display rows in the window). *Delim* must be specified.

exit *Exit* returns the ASCII code for the exit key used.

inpat Use *inpat* to specify input pattern data validation and conversion. Syntactically as well as operationally, this is exactly the same as an In Pattern operation in the Prompt window in Paint. Multiple *inpat* statements are processed as logical OR values and must be delimited by @VM or | (vertical bar). *Inpat* is applied to each line if *lines* specifies multi-line or single line scroll mode.

outpat Use *outpat* to specify output pattern conversion (applied line by line). Syntactically as well as operationally, this is exactly the same as an Out Pattern operation in the Prompt window in Paint (see the Paint documentation for more details).

just Use *just* to specify display line justification. The possible values are:

Value	Operation
L	Left justified
R	Right justified
C	Center justified
T	Text (text lines will wrap). *Delim* delimits each line.
TN	Text nonjustified (text lines will wrap).

delim Use *delim* to specify 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 Use *mask* to specify 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 Use *linelen* to specify 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 Use *state* to pass 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.INSERT        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 *inval* data (*flags*<SAVE.STATE$> must be false).
<SS.LAST.VAL>	Contents of current display line.
<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.INSERT>	Stores the current insert mode (true = insert, false = overstrike). The default insert mode is readout of @INSERT if the editor is toggled.
<SS.FIND>	Current search string.
<SS.REPLACE>	Current replacement string.

Field	Description
<SS.STATE>	Changes in edit status. Notice which values are 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 *inpat* operations (set by SCRIBE). **2** Process current row through *inpat* (passed to SCRIBE). **3** Process full edit session through *inpat* (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
inval	Edited *inval* data	None. Always returned.
key	The key pressed to exit SCRIBE	None. Always returned.
change	True or false (1 or 0)	None. Always returned.
state	Current edit session parameters	SCRIBE must have been called with *flags*<SAVE.STATE$> set to true.
	Specified edit session parameters	SCRIBE must have been called with *flags*<SAVE.STATE$> set to false and the specified edit session parameters set in *state*.
	The original (unedited)	SCRIBE must have been called with *flags*<NO.ORIG.VAL$> set to false and *state* 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 parameters. The program loops through and re-calls SCRIBE until the user chooses to quit. At each iteration, the user can reinitialize the data passed to SCRIBE.

```
$INSERT SYSINCLUDE, SCRIBE.FLAGS.CONSTANTS
$INSERT SYSINCLUDE, SCRIBE.STATE.CONSTANTS
$INSERT SYSINCLUDE, LOGICAL
$INSERT SYSINCLUDE, WINDOW.POINTERS

DECLARE SUBROUTINE MSG, SCRIBE, BORDER.UP
DECLARE SUBROUTINE INPUT.CHAR, DELAY
DECLARE FUNCTION VIDEO.RW


EQU null          TO ""
EQU F9            TO CHAR(0):CHAR(67)
EQU esc           TO CHAR(27)
EQU cursor.down   TO CHAR(0):CHAR(80)

verrors     = ""
verrors<1>  = "Out of memory during read attempt"
verrors<2>  = "Display is not memory-mapped"
verrors<3>  = "Col or Row is out of range"
verrors<4>  = "Left col is greater than right col, or"
verrors<4,1>   = "top row is greater than bottom row"
verrors<5>  = "Operator is not 'C', 'R', 'W'"
verrors<6>  = "Size of saved image is less than area ⇨
   to be restored"
* use VIDEO.RW to save and clear screen area
left     = 0
top      = 0
right    = @CRTWIDE-1
bottom   = @CRTHIGH-1
operator = ""
image    = ""
operator = "R"
GOSUB call_video.rw
image1 = image
;* preserve original screen contents

* set white on blue, with space background
attributes = \201F\
image      = attributes
operator   = "C"
OSUB call_video.rw
```

(continued)

```
* 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
* clear boxed area and put border around it
attributes = \B074\  ;* patterned, light red background
operator = "C"
image    = attributes
GOSUB call_video.rw
DELAY(2) ;* show the effects of the box clearing
BORDER.UP(left, top, right, bottom, 1, CURATR())
* add border
DELAY(2)    ;* show the effects of adding a border
* actual processing loop
done        = false  ;* flag for LOOP..REPEAT
first.time  = true   ;* flag for 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
      text = "Do you wish to call SCRIBE again?"
      MSG(text, "RC", resp, "")
      IF resp EQ "Y" THEN
* determine whether to re-initialize the default data
         redo = "N"
         text = "Re-initialize IVAL (Y/N)?"
         MSG(text, "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 */
```

(continued)

```
IF first.time THEN
* note use of @VM as delimiter (as in DELIM)
   ival = ""
   ival<1,-1> = "This is a"
   ival<1,-1> = "sample call"
   ival<1,-1> = "to SCRIBE"
END ;* if first.time
protect = false   ;* protects IVAL if true
exit_key = esc:@FM:F9   ;* keys that cause exit

* next line causes exit w/ editor OFF
except_key = cursor.down
col = left+1   ;* left column
row = top +1   ;* top row
displen = (right-left-2)   ;* display length

* basic text display, red on off-white
color = esc:'C':\0704\

* marked block, intense white on magenta
color<2> = esc:'C':\050F\

* current line, yellow on off-white
color<3> = esc:'C':\070E\

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
```

(continued)

```
   /* finally */
   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 and exit, restoring default values

exit:
left     = 0
top      = 0
right    = @CRTWIDE-1
bottom   = @CRTHIGH-1
image    = ""
operator = "W"
image    = image1 ;* restore original screen
* leave unconditionally
vstatus  = VIDEO.RW(left, top, right, bottom, ⇨
   operator, image)
STOP     ;* end of program

call_video.rw:
vstatus = VIDEO.RW(left, top, right, bottom, ⇨
   operator, image)
IF vstatus THEN
   MSG(verrors<vstatus>)
   GOSUB exit
END
RETURN
```

SET_ATTACH_IMAGE

This subroutine creates a record in the SYSENV file containing the current configuration of attached volumes and files.

```
SET_ATTACH_IMAGE ( name, batch_name, options, errors )
```

name — Pass in *name* a value to be the key to a record in the SYSENV file, for the attach configuration you are requesting.

batch_name — Pass in *batch_name* a value to be the key to a record in the VOC file. See the explanation under SETATTACH in the "TCL command reference" chapter of the *Reference* manual for details. You must manually create this record.

options — Pass "O" to give permission to overwrite any existing record having the key of *name*.

errors — The system returns in *errors* a variable that will receive any error code generated by this subroutine. SET_ATTACH_IMAGE returns null when successful. The following system message can appear upon failure:

Record key	Description
W109	Image record already exists

Using SET_ATTACH_IMAGE

The following code segment illustrates the proper use of SET_ATTACH_IMAGE:

```
SET_ATTACH_IMAGE( name, batch_name, options, errors )
IF LEN( errors ) THEN
   @FILE.ERROR = errors
   FSMSG()
END
```

See also USE_ATTACH_IMAGE (an R/BASIC system subroutine) and the SETATTACH TCL command.

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 is a single-line, three-region unit.

Region 1 Columns 1—66. Used to indicate information about the current process, and open to developers.

Region 2 Columns 68—76. Reserved for system use. "Indexing" appears here, when background indexing is executing. Column 67 contains a vertical bar separator.

When background indexing is not executing, the following series of flags displays:

CL	Caps Lock. Caps Lock mode is on.
NL	Num Lock. Num Lock mode is on.
OV/IN	Overstrike or Insert mode, for the editor. When the editor is off, neither symbol displays.
SEL	Select list is active.

Region 3 Columns 78—80. Reserved for system use, and contains the current window level counter.

For historical reasons, there are ten cell positions. Only cell 3, which displays as Region 1, is available for your programming use.

When certain programs run, it appears as if cell 3 comprises more than one 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 to indicate their status. This status line updating can take three forms:

- Display a pre-defined template
- Update one or more cells directly
- Display a template and update all or a portion of cell 3

operation

Use *operation* to specify 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 SYSINCLUDE 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 *data*.
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.
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

Use *location* to pass 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. You cannot use the PUSH$ operation and pass a null value if your Help level is 1 or 2 — you *must* pass a template name.
POP$	Unused (should be null).
SINGLE$	An integer indicating the status line cell 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 *data*. 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 *location*. 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 *location*.
MULTI.MODAL$	A dynamic array of data (delimited by | or @FM) to display the columns of cell 3 of the status line referenced by *location*.

Values returned

None.

Correct use of STATUP

```
$INSERT SYSINCLUDE, STATUS.CONSTANTS
DECLARE SUBROUTINE STATUP, MSG

/* Save the current status line state into variable
STATUP_STATE and display the status line INITIAL. which
is stored in the SYSTEXT file. */

MSG( 'Press a key to see the INITIAL status line')
STATUP(PUSH$, 'INITIAL', statup_state)

/* Display a message in the large modal cell of the
status line. */
MSG( 'Press a key to update the big (modal) cell.')
TEXT = "Processing - almost done..."
STATUP(SINGLE$, STAT.MODAL$, TEXT)

/* Now we are done and should restore the previous
status line state.  This state was stored in
STATUP_STATE at the beginning of this program. */

MSG('Press a key to restore the original status line')
STATUP(POP$, '', statup_state)

MSG('Done.')
```

USE_ATTACH_IMAGE

This subroutine attaches the system volume and file configuration that is specified in a particular SYSENV record.

```
USE_ATTACH_IMAGE( name, errors )
```

name Pass in *name* the image record you want to attach. Must be a valid record key in the SYSENV TABLE.

errors The system returns in *errors* any error codes resulting from this process. If *errors* returns null, the process was successful.

Record key	Description
202	Image record does not exist
B127	Not a valid image record
B308	Open media failed on volume %?%
201	Unable to open system file %?%

See the R/BASIC SET_ATTACH_IMAGE system subroutine for an example of usage.

User-Defined Conversions

In addition to standard data conversions, such as D (date) and MD2 (masked decimal), Advanced Revelation allows you to call custom conversion subroutines as data validation and conversion functions.

Using User-Defined Conversions

A user-defined conversion is an R/BASIC (or C or assembler) subroutine that contains the logic necessary to convert data into and out of a format required for an application. For example, you might want to convert Roman numerals into integers and vice versa.

You can use user-defined conversions in a variety of cases to convert or validate data in ways not predefined in Advanced Revelation:

- In a data entry window, by entering a user-defined conversion in the *Validation pattern* or *Output format* prompts of the Prompt window.
- When importing data, by entering a user-defined conversion in the *Validation pattern* or *Output format* prompts of the Dictionary window.

- When entering data into a file programmatically, by processing the data using the R/BASIC ICONV command and a user-defined conversion.

Note For more information on data validation, see "Data Validation" in the chapter "Creating windows with Paint" in the *User's Guide*.

When calling a user-defined conversion in an R/BASIC program, the call to I/OCONV must pass the name of a cataloged subroutine as the conversion argument. For example, to call a conversion subroutine called ROMAN, you might use the following statement:

```
X = OCONV(Data, "[ROMAN]" )
```

When calling a subroutine using the R/BASIC ICONV or OCONV commands, the subroutine name must be in uppercase, square brackets, and quotation marks. When calling a subroutine from the Validation Patterns or Output Format prompts of the Prompt window or the Dictionary window, do not include the quotation marks.

The following example calls the subroutine ROMAN, passing it a total of 4 arguments, the values for which are supplied by the system. The called subroutine must have a first line specifying these arguments (the variable names are not significant):

```
SUBROUTINE ROMAN(type, value, branch, return_value)
```

Once the subroutine has control of the data, it is the responsibility of the subroutine to convert the data, as well as to set flags for the success of the conversion.

The conversion subroutine can manipulate the input data in any way. You should be careful to remember, however, that whatever you do to the data in *value* will be reflected in the program that called the subroutine. It is wise to copy this data to *return_value* immediately, then manipulate *return_value* only.

The subroutine can use the STATUS() function to report whether a conversion has been successful. The conventional values for this function are:

```
STATUS() = 0  ;* conversion OK
STATUS() = 1  ;* data can't be converted as requested
STATUS() = 2  ;* invalid branch or conversion
STATUS() = 3  ;/* failed conversion; error handled by
                  the subroutine handling conversion*/
```

The subroutine itself need not act on any of these conditions, nor is it absolutely required to set the status flags. However, consistency with other Advanced Revelation functions dictates following the above conventions for status settings. See the "Error Conditions" topic in "Accessing Advanced Revelation tables" of the "Programming with R/BASIC" chapter.

When developing conversion subroutines that involve branching, it is important to ensure that the subroutine can handle both ICONV and OCONV input and that the input arguments be specified identically in each case. Certain internal processes require that input/output conversions go either way. Therefore, include processing for all branches under both the ICONV and OCONV sections of the conversion

subroutine. If you are using three degrees of branching and using "*" (asterisk) as a parsing symbol, your subroutine must handle the following calls:

```
ICONV(data, "[MYCONV, PARAM1*PARAM2*PARAM3]")
OCONV(data, "[MYCONV, PARAM1*PARAM2*PARAM3]")
```

```
custom_subroutine(type, value, branch, return_value)
```

type *Type* is used to specify whether this is an ICONV or OCONV. The word "ICONV" or "OCONV" is passed as a literal by the I/OCONV function itself (no programmer intervention required).

value *Value* is used to pass the data to be converted.

branch *Branch*, an optional argument, can be used for special processing. If a branch is desired, the extra argument is specified in the [ ] call in the CONV function itself, like this:

```
X = OCONV( DATA, "[ROMAN,3]" )
```

The optional branch argument can be alpha or numeric: it is simply passed through to the cataloged routine. Only one branch is permitted (but see the sample code below for ideas on overcoming this limitation).

return_value The variable *return_value* is passed to the custom subroutine to accept the results of the conversion.

Values returned

User-defined subroutines must return their result in *return_value*.

Correct use of User-Defined Conversions

Three routines below illustrate various aspects of user-defined conversions. The first routine sets up user-specified conversion situations. Depending upon the input request, the following two routines (which must have been cataloged before the first is run) could be invoked to handle specific types of conversions.

Two user responses will invoke the user-defined conversion subroutines: "[%]" invokes PERCENT (which must be cataloged with the name "%") and "[MDE, x/n]" invokes MDE, with branching requests specifying country and desired number of decimal places (see below for details).

```
* the primary conversion request routine:
DECLARE FUNCTION VIDEO.RW
DECLARE SUBROUTINE MSG

EQUATE esc$    TO @INT.CONST<1>
EQUATE tab$    TO "      "
EQUATE true$   TO 1
EQUATE false$  TO 0
EQUATE mjust$  TO 6  ;* MAP argument for MSG; field 6
                     ;* is justification
done  = false$
conv     = ""  ;* variable to hold type of conversion
data     = ""  ;* variable to hold data for conversion
header   = "Example conversion types:"

/* MSG won't display certain characters, so use the
%Cn% convention. %C37% is %, and %C124% is |. */

header<-1> = "D, MD2$, [%C37%], [MDE, (country) ⇨
   %C124%(#_of_decimal)]"
header<-1> = "  for COUNTRY, choose E, G, I, or F   "
header<-1> = ""
header<-1> = "Enter the type of conversion: "
map   = ""  ;* second MSG parameter
map<mjust$> = "L" * store and clear the screen
vresult = VIDEO.RW(0, 0, @CRTWIDE-1, @CRTHIGH-1, "R",⇨
   image)
IF vresult THEN STOP
PRINT @(-1)  ;* clear the screen

LOOP
   MSG(header ,"RCE", conv, "")
   IF conv = esc$ THEN done = true$
UNTIL DONE
   MSG("Enter data to be converted","RCE",data,"")
   IF data = esc$ THEN done = true$
UNTIL DONE
   GOSUB attach_values
   MSG(message, map,"","")
REPEAT

* restore the screen and exit

vresult = VIDEO.RW(0, 0, @CRTWIDE-1, @CRTHIGH-1, "W",⇨
   image)
STOP
```

(continued)

```
/ ****** Internal Subroutines ****** /
attach_values:
value1 = "Data to be converted: ":data
value2 = "Internal conversion:  ":iconv(data, conv)
value3 = "Output conversion:    ":oconv(data, conv)
message  = value1
message<-1> = value2
message<-1> = value3
message<-1> = ""
RETURN
```

The following subroutine handles percent conversions. This code must be separately compiled and cataloged.

```
SUBROUTINE PERCENT(type, value, branch, return_value)

/* Example of a user-defined conversion: converts from
decimal to percentage and back (nn% format). */
DECLARE SUBROUTINE MSG
return_value = value
STATUS() = 0
BEGIN CASE
   CASE type = "ICONV"
      GOSUB Iconv
   CASE type = "OCONV"
      GOSUB Oconv
   CASE 1
      return_value = ""
      STATUS() = 2
END CASE
RETURN

Iconv:
CONVERT "%" TO "" IN return_value
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 := "%C37%"
END ELSE
   return_value = ""
   STATUS() = 1
END
RETURN
```

VIDEO.RW

The function VIDEO.RW allows you to save, restore, and clear specific areas of the display.

```
error = VIDEO.RW(left, top, right, bottom, operator,image)
```

Using VIDEO.RW

VIDEO.RW addresses areas of the screen by indicating the left and right column numbers (0-79), and the top and bottom row numbers (depends on video mode). A program can use these four coordinates to create the dimensions of a box. The box can then be read, cleared, or written to, and then restored to its original state.

When you read part of the display screen using VIDEO.RW, you store that image in a variable. When you restore the screen image with a write, you use that variable to restore the data for the image. It is possible to restore a different image by using a different variable, but if the new data is too small for the area to be restored, VIDEO.RW will generate an error. If the data is too large, only the portion that fits will be displayed.

Typically, you use VIDEO.RW in a three-step process:

- Read the relevant display area (saving that image in a variable).
- Clear the screen within that area (optionally changing the background character and/or its attribute).
- When ready, write the image back from the stored variable.

left Use the *left* parameter to pass a value for the left column coordinate of the window space being defined. Be sure that this value is less than the value of *right*.

top Use the *top* parameter to pass a value for the top row coordinate of the window space being defined. This value must be less than that for *bottom*.

right Use the *right* parameter to pass a value for the right column coordinate of the window space being defined. This value must be greater than that for *left*.

bottom Use the *bottom* parameter to pass a value for the bottom row coordinate of the window space being defined. This value must be greater than that for *top*.

operator There are three possible actions to be taken by VIDEO.RW:

R Read and save the specified area of display.

W Write the stored image back to the specified area.

C Clear the specified area of display.

image If *operator* is "R", *image* is the variable to which the display is written.

If *operator* is "W", *image* is the variable from which the display is restored.

If *operator* is "C", *image* can be a two-byte value affecting the background and display attribute values. The first byte controls the background character, while the second byte controls the display attribute. For more information, see the WINDOW.POINTERS record in the SYSINCLUDE file.

Values returned

This function returns only one value:

error The value returned by this function is an error code:

Error code	Meaning
0	Operation successful
1	Out of memory during read attempt
2	Display is not memory-mapped
3	Column or row is out of range
4	*Left* exceeds *right* or *top* exceeds *bottom*
5	Argument for mode was not "C", "R", or "W"
6	Saved image is smaller than area to be restored

Correct use of VIDEO.RW

The following code demonstrates the three operator modes of VIDEO.RW. It first saves the display image of a central part of the screen, then clears it (substituting various video attributes), and finally restores the stored image.

```
DECLARE FUNCTION VIDEO.RW
DECLARE SUBROUTINE MSG, MONITOR
left     = 20
top      = 5
right    = 60
bottom   = 15
```

(continued)

```
operator = "R"
image    = ""
background  = CHAR(178)
monochrome  = CHAR(143)

bgrd_color  = 5      ;* dark magenta
frgr_color  = 14  ;* light yellow
color       = (bgrd_color * 16) + frgr_color
verrors  = ""
verrors<1>  = "Out of memory during read attempt"
verrors<2>  = "Display is not memory-mapped"
verrors<3>  = "Col or Row is out of range"
verrors<4>  = "Left col is greater than right col, or"
verrors<4,-1> = "top row is greater than bottom row"
verrors<5>  = "MODE is not 'C', 'R', 'W'"
verrors<6>  = "Size of saved image is less than ⇨
   area to be restored"

error = VIDEO.RW(left, top, right, bottom, operator,⇨
   saved_image)
IF error THEN
   MSG(verrors<error>)
   STOP
END
MONITOR(status)
operator = "C" ;* clear the screen

* a value of 7 indicates monochrome monitor
IF status = 7 THEN
   image = background:monochrome
END ELSE
   image = background:color
END
error = VIDEO.RW(left, top, right, bottom, ⇨
   operator, image)
IF error THEN
   MSG(verrors<error>)
   STOP
END

MSG("this is just to delay","T2")
operator = "W"
error = VIDEO.RW(left, top, right, bottom, ⇨
   operator, saved_image)
IF error THEN
   MSG(verrors<error>)
END
```

Index

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

RevelationTechnologies

181 Harbor Drive
Stamford CT 06902
(203) 973-1000
(800) 262-4747

Revelation Technologies (UK), Ltd.
270 Upper Fourth Street
Central Milton Keynes
MK9 1DP England
0908-233-255

300-2002-00