HIGH PERFORMANCE

LEVEL II COBOL™

OPERATING GUIDE

Version 2.0

HIGH PERFORMANCE

LEVEL II COBOL™

Operating Guide

(For use with the UNIX™

Operating System)

Version 2.0

Micro Focus Ltd

Issue 1
February 1984

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection herewith.

The authors and copyright holders of the copyrighted material used herein:

> FLOW-MATIC (Trademark for Sperry Rand Corporation) Programming for the Univac I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DS127A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Micro Focus has made every effort to ensure that this manual is correct and accurate, but reserves the right to make changes without notice at its sole discretion at any time.

The software described in this document is supplied under a license and may be used or copied only in accordance with the terms of such license, and in particular any warranty of fitness of Micro Focus software products for any particular purpose is expressly excluded and in no event will Micro Focus be liable for any consequential loss.

Note that the following are registered trademarks:

> LEVEL II COBOL™ (LEVEL II COBOL), FORMS-2™ (FORMS-2), and ANIMATOR™ (ANIMATOR) are trademarks of Micro Focus.
> UNIX is a trademark of Bell Laboratories.
> DEC is a trademark of Digital Equipment Corporation.
> ADM-3A is a trademark of Lear Siegler, Inc.
> DPPX is a trademark of the IBM Corporation.
> IBM is a trademark of the IBM Corporation.

© COPYRIGHT 1984 by Micro Focus Ltd.

LEVEL II COBOL OPERATING GUIDE

VERSION 2.0

AMENDMENT RECORD

Amendment Number	Dated	Inserted by	Signature	Date

PREFACE

This manual describes operating procedures for the UNIX resident releases of the LEVEL II COBOL Version 2.0 compiler and Run-Time System (RTS).

The compiler converts LEVEL II COBOL source code into an intermediate code. This intermediate code can then be interpreted by the Run-Time System. Alternatively, the intermediate code can be converted by the native code generator into the machine code for your processor. This native code can then be run under the control of the Run-Time System.

This manual describes the steps needed to compile a program, generate native code, and execute the intermediate or native code, including all necessary linkage and run-time requirements. Operation of the run-time interactive debug package is also described.

CONTENTS

This manual contains the following chapters and appendices:

"Chapter 10. Terminal Configuration Considerations", describes how the Run-Time System operates with various types of terminals.

"Chapter 11. Incorporating FORMS-2 Utility Program Output", describes the use of output from the FORMS-2 screen formatting utility program.

"Appendix A. Summary of Compiler Directives", summarises the compiler directives available for the LEVEL II COBOL compiler.

"Appendix B. Compile-Time Errors", lists the error numbers and descriptions reported by the compiler at compile time.

"Appendix C. Summary of Native Code Generator Directives", summarises the directives available for the native code generator.

"Appendix D. Code Generation Errors", lists the error numbers and descriptions reported by the native code generator during code generation.

"Appendix E. Run-Time Errors", lists the error numbers and descriptions reported by the Run-Time System during program execution.

"Appendix F. Operating System Errors", is a listing of the messages for errors originating in the UNIX operating system.

"Appendix G. Interactive Debug Commands", summarises the commands that can be used with the LEVEL II COBOL interactive debug option.

"Appendix H. Terminal Configuration Issues", describes how to add a new type of terminal to the configuration file.

"Appendix I. UC Berkeley Termcap", contains the documentation from UC Berkeley on the termcap terminal capability description file.

"Appendix J. LEVEL II COBOL in the UNIX Environment", summarises the difference betweeen this LEVEL II COBOL implementation and others with which you may be familiar.

"Appendix K. lockdemo Source Code", provides the source listing of lockdemo.cbl, the record locking demonstration program, which is explained in Chapter 9.

NOTATION IN THIS MANUAL

Throughout this manual, the following notation is used to describe the format of data input or output:

1. When material is enclosed in square brackets [ ], it is an indication that the material is an option which may be included or omitted as required.

2. The symbol << after a CRT entry or command format in this manual indicates that the CR (carriage return) or equivalent data input terminator key must be pressed to enter the command.

3. All numbers are in decimal unless otherwise stated.

Headings are presented in this manual in the following order of importance:

CHAPTER n
TITLE
} Chapter Heading

ORDER ONE HEADING
ORDER TWO HEADING
Order Three Heading
Order Four Heading
} Text 1 line down

Changes issued as part of an Addendum are identified by the addendum number printed at the bottom of the changed page. It is suggested that Addendum page change instructions are filed at the back of the manual as a detailed record. Page iii is provided as a record of all amendments to your copy of the manual.

Related Publications

For details of the LEVEL II COBOL language, refer to the document:

LEVEL II COBOL Language Reference Manual

For details of the UNIX operating system, error messages and file structures, refer to the UNIX Programmer's Manual.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

CHAPTER 2

COMPILER CONTROLS

CHAPTER 3

NATIVE CODE GENERATOR CONTROLS

CHAPTER 4

RUN-TIME SYSTEM CONTROLS

CHAPTER 5

INTERACTION IN APPLICATION PROGRAMS

CHAPTER 6

INTERACTIVE DEBUGGING

CHAPTER 7

MULTI-LANGUAGE CALL FACILITIES

CHAPTER 8

LEVEL II COBOL APPLICATION DESIGN CONSIDERATIONS

CHAPTER 9

FILE AND RECORD LOCKING FOR INDEXED FILES

CHAPTER 10

TERMINAL CONFIGURATION CONSIDERATIONS

CHAPTER 11

INCORPORATING FORMS-2 UTILITY PROGRAM OUTPUT

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

GENERAL DESCRIPTION

COBOL (COmmon Business Oriented Language) is the most widely and extensively used language for the programming of commercial and administrative data processing. LEVEL II COBOL is a compact interactive standard COBOL language system.

The LEVEL II COBOL compiler converts LEVEL II COBOL source code into an intermediate code. This intermediate code may be submitted as input to the native code generator, which will generate the equivalent machine code for your processor type. Either the intermediate code or the native code is then executed by the Run-Time System (RTS).

LEVEL II COBOL programs can be created by using the standard UNIX text editor to produce LEVEL II COBOL source files. The compiler translates these source files into intermediate code files. These files consist of instructions for an abstract "virtual machine" implemented by the RTS. Thus, intermediate-code files specify operations that the machine-specific RTS interprets and implements.

A listing of the LEVEL II COBOL program is provided during compilation, including any error messages produced by the compiler. The listing also provides addresses for use with the interactive debug utility (see Chapter 6). To execute the program after compilation is finished, invoke the RTS by using the cbrun command followed by the intermediate code file name as an argument.

You can often considerably improve the run-time speed of your programs by using the native code generator to turn your intermediate code into the machine code specific to your processor. The native code generator will provide an assembly listing of the generated code that it produces, for documentary purposes. You execute a generated code program in exactly the same way as an intermediate code program, using cbrun followed by the name of the file containing the generated code.

Generated code is not necessarily always preferable to intermediate code. Chapter 8, APPLICATION DESIGN CONSIDERATIONS, includes a brief discussion on when it is worth while producing a generated code version of your program. The section PROGRAM DEVELOPMENT CYCLE, later in this chapter, shows at what point you should use the native code generator.

The LEVEL II COBOL system also provides a powerful utility program called FORMS-2. FORMS-2 allows the user to define the screen layouts to be used in a LEVEL II COBOL application. Screen layouts are created by simply typing them in on the CRT. FORMS-2 can also be used to automatically generate a program that can create and maintain data files based on these screen formats.

The RTS executes either the intermediate code file produced by the compiler or the generated code file produced by the native code generator. In addition to standard ANSI COBOL statements, LEVEL II COBOL provides many extensions for interactive programs. The RTS requires knowledge of the terminal type for these features, which is obtained by use of the termcap configuration file and associated utilities.

If your terminal is not listed in the configuration file provided, see Chapter 10 and Appendices H and I for information on how to include it. In order for the RTS to access the terminal type, the TERM shell variable must be set to the terminal code. If the Bourne shell is in use, it should be placed in the environment by use of the "export" command. Again, see Chapter 10 and Appendices H and I for more information. Additional information can be obtained from the UNIX Programmer's Manual.

THE DEMONSTRATION PROGRAMS

Three demonstration programs, pi.cbl, stock1.cbl and stock2.cbl, are supplied in source form. They may be used to familiarize oneself with the system. A multi-user test program for the Indexed Sequential Access Method (ISAM) is also supplied.

GETTING STARTED

COMPILATION

The first thing to do is to compile all of the demonstration programs. These are the files with the extension ".cbl". Your current directory should contain these files. If not, you will need to provide the full path name to these files wherever the file name is required.

EXAMPLE:

```
cobol stock1.cbl<<
```

When compilation completes, the output should be:

```
* LEVEL II COBOL V2.0  Copyright (c) 1982 Micro Focus Ltd
* Compiling stock1.cbl
* ERRORS=00000 DATA=01024 CODE=00512 DICT=04949:21506/26455 GSA FLAGS=OFF
```

A listing of the current directory will show that two new files exist: stock1.LST, which is the list file, and stock1.INT, which contains the intermediate code. The programs stock2.cbl and pi.cbl can be compiled in the same manner.

When stock2 is compiled you will notice three additional lines of output to the terminal, which indicate that an error was detected in the program. The message produced by the error is:

```
          MOVE GET-INPUT TO TF-DATE.
**103*********************                                    ******
**     Operand has wrong data-type, is not declared or "." missing     ******
```

The first line is a copy of the line where the error was detected. The second line contains the error number. The third line is the error message. A complete listing of error numbers and messages can be found in Appendix B. The error above was deliberately included in the program to provide a demonstration of the compiler's error message format, and does not interfere with the correct operation of the rest of the program. Note that compilation errors do not prevent the creation of an intermediate code file. The offending statements produce no-ops, that is, the operation code for "do nothing and continue". The intermediate code file will generally execute, though probably not correctly if errors are present. You are advised not to run intermediate code produced by a compilation with errors.

CODE GENERATION

If you wish, you can take the further step of generating native code versions of the demonstration programs. For example:

```
cgen pi.INT
```

will take the intermediate code produced by the compiler for pi.cbl and produce the equivalent machine code program. The .INT extension is not strictly necessary, since the native code generator will automatically append this extension if it is missing.

When code generation is complete, a listing of your current directory should show the presence of the file pi.GNT, which contains the generated code for pi.INT.

EXECUTION

To execute a program, type:

```
cbrun
```

with the appropriate file name as an argument. The file may be an intermediate code file (extension .INT) or a generated code file (extension .GNT). If there is no extension, the run-time system will first search for the file itself, then the file name with .GNT appended, then the file name with .INT appended.

The following programs have been included to provide a demonstration of the LEVEL II COBOL system.

Calculate the Value of PI

```
cbrun pi<<      ( or  cbrun pi.INT<<  or cbrun pi.GNT<<)
```

This clears the screen and displays the lines below with different initial numerical values. These are then overwritten with each iteration of the loop that calculates pi.

```
CALCULATION OF PI

NEXT TERM IS 0.000000000000

   PI IS  3.141592653589
```

The execution of the program terminates when NEXT TERM reaches zero.

Note that the RTS, in order to execute pi, must be aware of certain features of the particular terminal being used. Terminal features are used in a file named "termcap", which the RTS refers to when executing pi. This is described more fully in Chapter 10, and Appendices H and I. Briefly, however, if the RTS responds with an error number 191 when "cbrun pi" is attempted, then either: 1) the TERM variable is not set and is not present in the "environment" (for example, the appropriate shell commands for the Bourne shell might be "TERM=adm31; export TERM"), or 2) the terminal type is not defined in termcap. If the RTS responds with error number 192, this means that the complete set of required capabilities is not defined in termcap for your terminal.

Stock Control Program 1

To run the stock control program, type:

```
cbrun stock1<<
```

This clears the screen, followed by -

```
STOCK CODE        <    >
DESCRIPTION       <                        >
UNIT SIZE         <    >
```

This is a skeleton data entry program in which stock records are created on a file in stock code order.

It also allows you to check out the cursor control and input validation functions. Refer to Table 5-1, Cursor and Function Control Keys, for the cursor control keys on some common terminals.

Using the cursor control keys you may move forward and backward from one data input field to the next, or forward and backward non-destructively one character position at a time within input fields. You may also move the cursor directly to the first character position in the first data input field, usually with a HOME key. You should note that the RETURN key, or other equivalent, is used to enter the completed data record (which consists of all of the fields displayed) and not the cursor control keys.

In this example the STOCK CODE and DESCRIPTION fields are alphanumeric (PIC X), which means that they will accept any printable character that is typed in. UNIT SIZE on the other hand, is a numeric field (PIC 9), and only digits are accepted. Consequently, the entire field must be filled by numerics, i.e. all " " (space) characters must be written over by numerics. Otherwise the input will be rejected when RETURN is pressed. Leading zeros must therefore be entered for smaller numbers, or fields may be automatically justified using the "Left Zero" cursor control key (usually the decimal point).

This program also creates an indexed sequential file called STOCK.IT, together with its index called STOCK.IT.idx.

To create a record, key the data into the unprotected fields delimited by "<" and ">". When a record is complete, press the RETURN key and the record will be entered in the data file. The unprotected areas will then be space filled ready for the next record to be entered if no errors are detected by the program. If the record remains displayed, the record was incorrectly keyed and must be corrected. This is done by using the cursor controls to move to the position of the error and typing over the existing characters with the correction. Pressing RETURN again will result in the record being accepted if there are no remaining detectable errors.

The execution of the program is terminated by pressing RETURN with the STOCK CODE field left blank. The program will then respond with "END OF PROGRAM" and terminate.

Stock Control Program 2

```
cbrun stock2<<
```

This clears the screen followed by -

```
          GOODS INWARD
STOCK CODE       <    >
ORDER NO       <      >
DELIVERY DATE MM/DD/YY
NO OF UNITS      <    >
```

This is a skeleton stock data input program in which the stock records created by stock1 can be accessed. The program updates the inventory record by entering goods received. It also creates a transaction file with the name STOCK.TRS.

The same cursor control features are present as in stock1. Note that the DELIVERY DATE has a different method of prompting than has so far been used, where the values are entered by typing over the letters with the indicated information.

If the input fields are accepted and if the STOCK CODE corresponds to one in the file created by stock1, then the relevant information from that file will be displayed along with a request for a yes or no response as to whether the information is correct and should be entered in the files. Any response other than "Y" (or "y") will cause the program to discard the update and start over again.

The program is terminated by entering all spaces for the stock code in the same way as for the stock1 program.

PROGRAM DEVELOPMENT CYCLE

Figure 1-1 shows the typical development cycle for a LEVEL II COBOL program. There are two distinct phases in the cycle:

1. The testing phase. In this phase, you compile your original COBOL source, editing and re-compiling as necessary to remove syntax errors. You then run the intermediate code program with various sets of test data to demonstrate the logical correctness of your program. At this stage you will probably use an interactive debugging tool to assist you in locating and correcting logic errors; this may be the LEVEL II COBOL interactive debugging facility described in Chapter 6, or the ANIMATOR debugging tool (provided it is available on your implementation).

2. The production phase. When you are satisfied that your program is logically correct, you have two alternatives:

   * Use the tested intermediate code program as the final working program.

   * If you decide, on the basis of the considerations set out in Chapter 8, that your program will run more efficiently as native machine code, you can submit the tested intermediate code to the native code generator, and use the resulting generated code as the final working program.

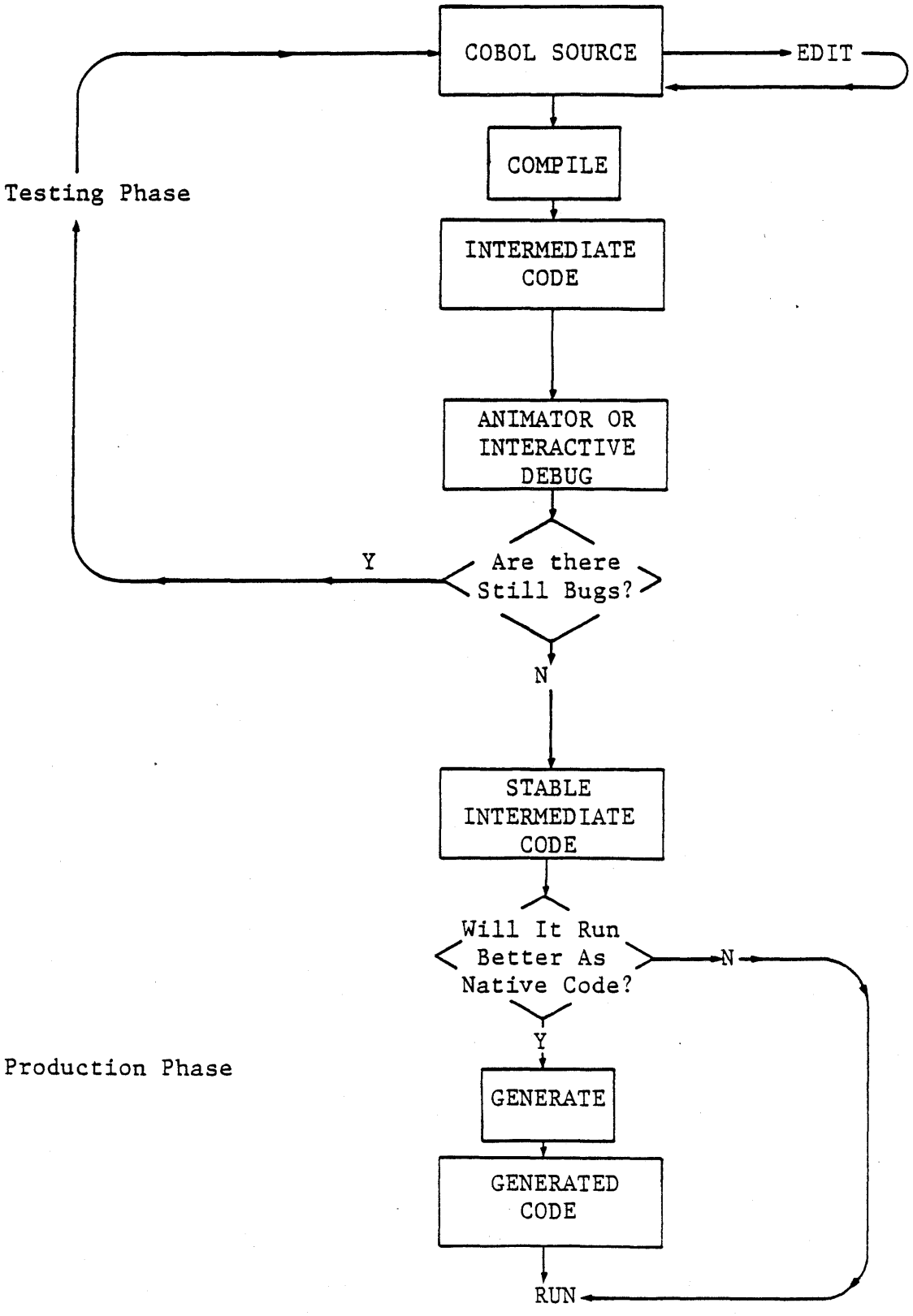
PROGRAM PREPARATION CONSIDERATIONS

The following should be noted:

1. The compiler rejects most non-alphanumeric characters within the source-code file (e.g. the Tab character) unless these are embedded in literal strings.

2. Lower-case characters in source-code files are generally accepted by the compiler. This is because the compiler considers lower and upper case characters to be identical in most cases. So, for example, a variable can be defined with upper case letters and can subsequently be used throughout the program in either upper or lower case. See Appendix J for more information.

Figure 1-1. A Typical Development Cycle.



CHAPTER 2

COMPILER CONTROLS

COMMAND LINE SYNTAX

The operation of the LEVEL II COBOL compiler is controlled by directives, most of which control the listing format. The compiler requires that these directives (such as NOLIST) be placed after the source file name on the command line. As a convenience to UNIX users, an alternative syntax is provided. This alternative allows short, dash-prefixed single character option flags to represent some of the standard directives. The "cobol" command translates these flags into the standard directive format before actually executing the compiler.

Either of the following command line formats may be used:

```
cobol prog.cbl  [ options ]  [ directives ]<<
cobol [ options ]  prog.cbl  [ directives ]<<
```

In these examples:

"prog.cbl" should be the name of a program file which contains LEVEL II COBOL statements. NOTE: file names beginning with a "." (for example, the path specification "../prog.cbl") may be used here. After compilation, the intermediate code file will reside in the same directory as the source code file. The ".cbl" suffix is optional but strongly recommended for the names of files containing COBOL source code.

"[options]" is a sequence of zero or more flags, each equivalent to a directive. Note that some directives do not have an option flag equivalent. Options are described alongside the directives to which they correspond.

"[directives]" consists of a sequence consisting of zero or more LEVEL II COBOL directives. The details of the input format for these directives are described in detail below, and the available directives are listed in the subsequent section.

DIRECTIVE AND OPTION INPUT FORMAT

The UNIX style "-" prefixed command flags may be used anywhere in the command line, and more than one option specification character may be placed after a single "-". For example, "-fc" is equivalent to "-f -c". Where, in the list below, there is an available UNIX-style command flag it is shown beneath the COBOL-style directive, with its equivalent meaning, e.g:

-c = COPYLIST

means the command flag -c is equivalent to the COBOL-style directive COPYLIST.

Each COBOL-style compiler directive must be separated by one or more spaces.

Where directives have parentheses, the left-hand parenthesis may occur any number of spaces after the body of the directive. Also, spaces are allowed within the parentheses. Parentheses in a command line create an additional problem in a UNIX environment, since they must be quoted or escaped to prevent their interpretation by the shell. This is shown in the following examples, all of which are acceptable and equivalent.

```
cobol prog.cbl "DATE(7-JAN)"
cobol prog.cbl DATE\(7-JAN\)
cobol prog.cbl DATE"(7-JAN)"
```

COMPILER DIRECTIVES LIST

A condensed version of this list is provided in Appendix A for convenient reference.

.	Returns control to the operating system, i.e. cancels this command sequence. Note that this must be quoted to avoid interpretation by the shell.
"&" or \&	Continuation character, if followed by a carriage return this permits the command sequence to continue on the following line. Note that individual directives may not be split over more than one line. Each line of directives will be accepted or rejected before the compiler allows the input of the continuation line.
[NO] ALTER	NO ALTER prohibits the use of ALTER statements within the program being compiled. This allows the compiler to operate more efficiently. The default is ALTER.

[NO] ANIM

Causes the program to be compiled in a manner suitable for later animation. See the manual LEVEL II COBOL ANIMATOR Operating Guide for more details. This directive is useful only if ANIMATOR is available with this implementation.

By default this directive is off unless ANIMATOR is available, in which case it is on.

[NO] BRIEF

Error numbers only are produced on the listing and console, i.e. the text of error messages is suppressed.

By default this directive is off unless no error message file can be found.

[NO] COMP

Causes the compiler to generate much more compact and efficient code for certain statements involving PIC 9(2) COMP and PIC 9(4) COMP data items. See Chapter 8 for full details. The reason for this directive is that the efficient code leads to non-standard behaviour in cases of numeric overflow; the compiler cannot allow this to happen unless you give this directive, meaning either that you know your statements will not lead to numeric overflow (in which case the semantics of your program will remain strictly in accord with the ANSI standard, while at the same time giving you the advantage of the extra efficiency), or alternatively that you mean to take advantage of the defined but non-standard behaviour on overflow.

By default this directive is off.

[NO] COPYLIST ["integer"]

-c = COPYLIST

Causes the contents of any files named in COPY statements to be listed.

By default this directive is off.

Whatever the state of this directive, the name of any copy file open at the time a page heading is output will be listed as part of the heading.

The optional integer, which must be zero or 50-99, allows the selection of particular segments with this directive. Zero means all root segments. For example:

COPYLIST "53" causes COPYLIST to be set in the identification division and in Segment 53 but not otherwise.

NO COPYLIST "53" causes COPYLIST to be set in Segment 53 only.

No integer may be specified if the -c notation is used.

[NO CRTWIDTH
 CRTWIDTH "integer"]

Specifies the width of the user screen in characters. This is used in Format 1 (standard ANSI) DISPLAY statements to enable the user to plan the separation points in display of data-items too long to fit on one physical CRT line.

By default this directive is set to 128.

Turning the directive off (NO CRTWIDTH) causes Format 1 DISPLAY statements to be rejected. Not using DISPLAY statements saves space in memory due to the control tables not being required.

DATE "string"

Causes the "string" to be used in place of the comment entry in the DATE-COMPILED paragraph (if present). Specifying NO DATE causes spaces to be used.

If the directive is omitted the comment entry (if present) is used.

[NO] ECHO

-k = NOECHO

Causes error lines and flags to be echoed to the console. Each error will result in the source line producing it, the error number and (unless BRIEF is set) an explanatory message, being printed on the console.

By default, this directive is on.

[NO] ERRLIST

-e = ERRLIST

Causes the listing to be restricted to those COBOL lines containing syntax errors or flags, together with associated error messages.

By default this directive is off, i.e. a full list is produced.

FILESHARE

Enables specification of file and record locking syntax for indexed files (see Chapter 7).

If this directive is specified the following directives are also available:

COMMIT(INDEXED)	changes the default file-type for indexed files to committable.
RESTRICT(INDEXED)	changes the default file-type for indexed files to restricted.

```
[ NO   FLAG                    ]
[      FLAG "( LOW  )"         ]
[           ( L-I   )          ]
[           ( H-I   )          ]
[           ( HIGH  )          ]
[           ( L/II  )          ]
[           ( IBM   )          ]
```

Causes the output of GSA compiler certification flags during compilation for all features higher than the specified level:

LOW	-	GSA Low-level
L-I	-	GSA Low-Intermediate-level
H-I	-	GSA High-Intermediate-level
HIGH	-	GSA High-level
L/II	-	LEVEL II COBOL extensions to ANSI COBOL standard X3.23 1974 (see the LEVEL II COBOL Language Reference Manual).
IBM	-	IBM-compatible non-standard COBOL (see the LEVEL II COBOL Language Reference Manual, Appendix J).

By default this directive is off.

```
[NO   FORM                ]
[     FORM "integer"      ]
```

-f = NOFORM

Specifies the number of lines per page of the listing. "integer" must be at least 3.

By default, 60 lines are printed per page.

One form-feed character is always produced at the head of the listing file but if NO FORM is used no further form-feed characters and no page headings are produced in the body of the listing.

If the listing is directed to the console (by use of the LIST directive) then the first form-feed character is replaced by a blank line.

```
[NO   INT              ]
[     INT "filename"   ]
```

-i = NOINT

Specifies the file to be used to hold the intermediate code output by the compiler; if the specified file exists, it will be overwritten.

NO INT (or -i) suppresses the production of an intermediate code file (i.e. the compiler is used for syntax checking only).

By default the compiler adds .INT to the source-file name, replacing any existing file-name extension.

Note that if "filename" is specified without a "." suffix, then a suffix must be added later, or the RTS will be unable to find the file, and will respond with an error. The Run-Time System (RTS) assumes a file has the suffix .INT.

```
[NO  {LIST }                    ]
[    {PRINT}                    ]
[                               ]
[    {LIST }{"destination"}     ]
[    {PRINT}                    ]
```

-l = LIST ":CO:"
-n = NOLIST

Specifies the destination of the listing file, if an existing file is specified, it will be overwritten. The destination may be omitted, in which case the listing is directed to standard output.

NO {LIST / PRINT} suppresses the production of a listing.

If "destination" is ":CO:" (or -l is specified) then the listing is directed to the standard output - usually your terminal; if it is ":LP:", the listing is directed to a file on disk, which may be printed off if required.

If no directive is specified, the compiler forms a file name by adding .LST to the source-file name. If a directive is specified with no filename, the console will be used.

[NO] QUAL — NO QUAL prohibits qualified data-names or procedure-names in the program being compiled. This allows the compiler to operate more effectively.

The default is QUAL.

[NO] REF — Causes four-digit location addresses to be included on the right hand side of the listing file. Note that a listing with location addresses may be required in order to identify the locations reported in RTS error messages.

By default this directive is off.

[NO] RESEQ

-r = RESEQ — Causes the compiler to generate COBOL line sequence numbers, starting at 10 in increments of 10.

By default this directive is off.

EXCLUDED COMBINATIONS

Certain of these directives may not be used in combination. Table 2-1 shows the directives that are excluded if the directive shown adjacent in the left hand column is specified.

Table 2-1. Excluded Combinations of Directives

DIRECTIVE	EXCLUDED DIRECTIVES
ANIM	NOCRTWIDTH
NOLIST	LIST PRINT [NO]FORM RESEQ COPYLIST ERRLIST [NO]REF
ERRLIST	RESEQ COPYLIST [NO]REF

COMPILER COMMAND LINE EXAMPLES

The following command line is an example of the standard LEVEL II COBOL format:

```
cobol prog.cbl NOREF NOFORM RESEQ<<
```

This will cause a listing file to be created with sequence numbers, and no addresses or page headers in the listing. It will also create a line-numbered version of the source file. This output format will allow the listing to be used as the source file for the next attempt to compile the program, after any errors and error messages are edited out, and the three trailing lines at the end of the prog.LST file are deleted.

The following is an example of the alternative cobol command line format:

```
cobol -i  prog.cbl<<
```

The -i option causes the source file to be checked for syntax errors only; no intermediate code is generated.

DICTIONARY CONTROL

The LEVEL II COBOL compiler uses a virtual memory mechanism for dictionary control (symbol table) handling. The RTS creates a dictionary file called "/tmp/cobdNNNNN", where NNNNN is the process I.D. Individual blocks are buffered in memory as needed. The buffer area is usually resident in dynamically allocated memory, but this can vary depending on the type of installation.

The "-d" flag, which affects certain internal parameters, must be passed to the RTS for correct execution of the compiler. It is automatically provided in cobol.c, the C program used to initiate compilation.

The virtual dictionary size defaults to 30,000 bytes, and has a maximum limit of 65,000. It may be set, for example, to 60,000 with a flag to the RTS of the form "-dv60000". Note that this flag may be passed to the RTS from the cobol command line.

SUMMARY INFORMATION ON STANDARD OUTPUT

The compiler's initial response to an input command line should be:

```
* LEVEL II COBOL V2.0 Copyright (c) 1982 Micro Focus Ltd
```

Each directive is then acknowledged by the compiler on a separate line, and is either ACCEPTED or REJECTED. After all the directives have been acknowledged, the compiler opens its files and starts to compile. At this point it displays the message:

```
    * Compiling prog.cbl
```

If any file fails to open correctly, the compiler displays:

```
Open fail : <filename>
```

The compilation is aborted, returning control to the operating system. "Open fail" results, for example, if the source file is located in another directory, or if the file name was typed incorrectly.

For each error that occurs during compilation, the line is normally displayed along with an error number and an error message (a list of compile-time errors can be found in Appendix B).

When the compilation is complete the compiler displays the message:

```
**ERRORS=nnnn DATA=nnnn CODE=nnnn DICT=used:free/total GSA FLAGS= nnn
```

where:

ERRORS	-	denotes the number of errors found
DATA	-	denotes the size of the data area of the generated program
CODE	-	denotes the size of the code area of the generated program
DICT	-	denotes the number of bytes used, the number remaining free in the data dictionary, and the total (i.e. DICT=used:free/total)
GSA FLAGS	-	if FLAG(level) is specified, this indicates the number of features used which were not allowed at the given GSA level.

LISTING FORMATS

The general layout of the list file is as follows:

```
* LEVEL II COBOL V2.0              <filename>              PAGE:  nnnn
* 000001 statement 1
     .
     .
     .
  00000n statement n

* LEVEL II COBOL V2.0 REVISION n                          URN AA/0000/AA
* Compiler Copyright (c) 1982 Micro Focus Ltd
* ERRORS=nnnn DATA=nnnn CODE=nnnn DICT=used:free/total GSA FLAGS=  OFF
```

The first two lines of title information are repeated for each page. The final line is the same as on the terminal.

Note that, if you specify the REF directive during compilation, a hexadecimal value denoting the address of each dataname or procedure statement appears to the right of the page. Addresses of datanames are relative to the start of the data area, while addresses of procedure statements are relative to the start of the code area. There is some overhead at the start of the data area, and a few bytes of initialization code at the start of the procedure area for each SELECT statement.

CHAPTER 3

NATIVE CODE GENERATOR CONTROLS

COMMAND LINE SYNTAX

The command line, the means by which you invoke the native code generator, specifies the intermediate code file to be used as generator input, and the way in which the code generator processes the intermediate code file.

The code generator command line format is:

```
cgen prog.INT [options] [directives] <<
```

where

prog.INT is the name of the input (intermediate code) file,

and

directives is an optional sequence of code generator directives. Each directive must be separated by one or more spaces. If the command line is too long to fit on a single line, it may be continued by typing an ampersand "&" followed by carriage return and continuing the command on a further line. For those directives that take parameters in brackets, the left bracket may occur after zero, one or more spaces following the directive name.

As with the compiler directives, some of the code generator directives have UNIX style option equivalents (see the next section).

Note that if you do not include the .INT extension in the input file name, the code generator will append it automatically.

CODE GENERATOR DIRECTIVES

Round brackets or double quotes are used to delimit the parameters of some directives. For parameters in quotes, the content of the parameter is retained fully. Brackets, quotes, and the & directive will need to be escaped to avoid premature evaluation by the shell, as with compiler directives (see Chapter 2).

The available code generator directives are:

[NO] ASM

-a=ASM Specifies whether or not you require an assembly listing. The code generator listing file contains full details of all generated code in mnemonic form. If NOASM is specified with LIST, it contains only the code generator identity, command line details and summary statistics.

By default this directive is off.

[NO BELL
 BELL "integer"]

Defines the character used to cause the "bell" (i.e. the terminal's audible warning) to sound. "integer" is the ASCII character in decimal.

The default is 07.

Tunring the directive off (NOBELL or BELL "0" causes no bell character to be set).

[NO] CHECK

-c= NOCHECK

Specifies checking of run-time limit violations e.g. PERFORM stack, table bounds.

NO CHECK suppresses such run-time checks.

If no directive is specified, checking takes place.

[NO FORM
 FORM "integer"]
-f= NOFORM

Specifies the number of lines per page of the listing. "integer" must be at least 3.

By default, 60 lines are printed per page.

A form-feed character is always produced at the head of the listing file unless NO FORM is used. NO FORM specifies that no form-feed characters or page headings are produced anywhere in the listing.

If the listing is directed to standard output (by use of the LIST directive), interpretation of the form-feed character is dependent on your particular CRT.

[NO] GNT[(external-file-name)
 "external-file-name"]

-g= NOGNT

Specifies the file to which the generated code program is directed, if the file exists, it will be overwritten.

NO GNT suppresses the generation of an output file (i.e., the code generator is used for assembly code listing generation only).

By default the code generator adds .GNT to the source-file name, replacing any existing file-name extension.

[[NO] LIST
     LIST[(destination)
         "destination"]]

-l= LIST":CO:"
-n= NOLIST

Specifies the file to which the code generator listing is to be directed, if the file exists it will be overwritten.

The destination may be a printing device, i.e., the printer or the console.

NO LIST suppresses the production of a listing.

If no directive is specified, the code generator forms a file name by adding .GRP to the source file name. If a directive is specified with no file name, the console is used.

PAGETHROW "integer"

Specifies the ASCII character code for physical page throw on the printing device. The character code is expressed in decimal.

The default is 12.

GENERATING SEGMENTED COBOL PROGRAMS

Segmented COBOL Programs are compiled into a root intermediate code file (usually taking the default .INT file extension) and a further intermediate code file for each overlay segment. There is also an inter-segment reference (ISR) file. The overlay segments have file extensions from .I50 to .I99, corresponding to the COBOL segment numbers, and the ISR file has the extension .ISR.

When a segmented program is generated only the name of the root segment intermediate code file is specified in the command line. The code generator uses this name and establishes from the ISR file which overlay segments are present and generates all the segments in a single run. The output files mirror the input files: a root (.GNT) file, overlays with extensions from .G50 to .G99, and a generated inter-segment reference file with extension .GSR.

Each COBOL segment appears separately in the code generator listing file.

If further overlaying is required due to memory limitations more files are generated. See Chapter 8.

CHAPTER 4

RUN-TIME SYSTEM CONTROLS

COMMAND LINE SYNTAX

The Run-Time System (RTS) is invoked and controlled by means of a command line.

The command line syntax for running a LEVEL II COBOL object program is as follows:

```
cbrun [debug-param] [switch-param]  prog  [prog-params]<<
```

In this example:

"prog" is the name of the intermediate or generated code file. If you specify no extension, the RTS treats the file name as follows:

1. It searches for the named file (that is, the file without any extension).

2. If this fails, the RTS appends .GNT to the file name and searches for the corresponding generated code file.

3. If this fails, the RTS appends .INT to the file name and searches for the corresponding intermediate code file.

If you do specify an extension, the RTS will immediately search for the appropriate file.

"[debug-param]" consists of "+D" for the LEVEL II COBOL interactive debugger. This may only be specified if the program is an intermediate code program.

"[switch-params]" may be used to control program options.

"[prog-params]" are string values or file names to be used by the program as parameters.

All arguments must be separated by spaces.

These command line options are explained further in the following sections.

THE DEBUG PARAMETER

The optional debug parameter +D may be used to invoke the LEVEL II COBOL interactive debug package, which is documented in Chapter 6. This is distinct from ANSI debug (+DB), which is considered a switch parameter, and is batch-oriented.

Interactive debug is not supported for generated code.

THE SWITCH PARAMETERS

LEVEL II COBOL includes the facility of controlling events in a program at run time depending on whether or not you set programmable switches, as described in the SPECIAL-NAMES section in the LEVEL II COBOL Language Reference Manual. You set these switches at run time by use of the switch parameter to the cbrun command.

A switch parameter is either of the following:

+DB	:	Turns on the ANSI COBOL debug facility.
[+/-]N	:	N is an integer in the range 0-7. '+' turns on the switch, and '-' turns off the switch; the default is that all switches are off. These switches can be specified in any order, and the last appearance of any specific number takes precedence.

See examples later in this chapter.

THE PROGRAM PARAMETERS

These are any additional parameters required by the program, frequently the names of files to be accessed. Parameters can be accessed from the command line in either of two ways:

1. If the COBOL program opens the file name ":CI:" for reading, and this file is specified ORGANIZATION LINE SEQUENTIAL, then the first READ from this file will access command line parameters.

2. ACCEPT FROM CONSOLE also reads from the ":CI:" device, so the first ACCEPT FROM CONSOLE will access parameters. Note that ACCEPT without a FROM clause is by default ACCEPT FROM CONSOLE, unless CONSOLE IS CRT is specified in the SPECIAL-NAMES section.

ACCEPT FROM CRT will not access command line parameters. Pay special attention to ACCEPTs without a FROM clause, since they may or may not access command line parameters, depending on the inclusion in the program of CONSOLE IS CRT.

RUN-TIME FLAGS

These are flags passed to the COBOL RTS at run time.

Exclusive Indexed Sequential Access Method (ISAM) Flag

The '-S' or 'single user' flag is intended primarily for systems using creation locking. It is a run time flag that reduces the number and frequency of disk accesses required for ISAM locking. This is accomplished by locking a file at the first ISAM request and unlocking it only after the file is closed. Note that, after the lock is set, requests from outside processes will 'sleep' until the lock is removed or an interrupt is issued. No diagnostic is given as to the lock status. Also, care should be taken to remove the lock file in '/isam' if it remains due to premature program termination (See Chapter 9).

The intent is that users who are certain that no other process will be accessing their data files can get quicker ISAM response. However, the files are in fact locked and protected from other processes which are following the correct locking protocol.

It is possible to specify the method of file locking with the 'LOCK MODE IS' phrase listed in the source program (see the LEVEL II COBOL Language Reference Manual for more information). The '-S' flag will override such usage.

Memory Allocation (-m) Flag

The RTS requires a data area procured by use of the sbrk(II) routine. Normally the RTS grows to use the maximum data address space available, up to about 64K. There is, however, a command line flag to limit this request:

```
cbrun -mNNNNN prog
```

where NNNNN is a decimal number greater than about 8000, which corresponds to the highest address at which to try to set the "break". If the sbrk() call fails, it is attempted at least once at a lower position. This flag would be useful in a loaded or limited-memory environment when it is known that less memory is needed (i.e. small programs). If small programs are the rule, an installation should consider adding this flag as a default in the cbrun and cobol programs. Users could still get the maximum if their use of "-m" were the last one on the final command line. Caution should be exercised; if the break is set too low, some peculiar behaviour may occur.

COMMAND LINE EXAMPLES

```
cbrun +D +1 +2 +3 prog.GNT<<
```

This loads the generated code of the program "prog" with interactive debugging, and the programmable switches 1, 2 and 3 set.

A debug initial display appears on the terminal.

```
cbrun prog.INT 1 2<<
```

This loads the program "prog" from the intermediate file produced by the compiler and passes the user program parameters 1 and 2 to the program prog. Debug is omitted.

```
cbrun -2 +5 -7 +7 +DB  prog.INT<<
```

This loads the program "prog" from the intermediate file produced by the compiler, with programmable switches 5 and 7 on and 2 off.
Note that the last setting of switch 7 is accepted.
Switches 1, 3, 4, and 6 are off by default.
The ANSI debug run-time switch is also set.

CHAPTER 5

INTERACTION IN APPLICATION PROGRAMS

CRT SCREEN HANDLING

COBOL is traditionally a batch processing language; LEVEL II COBOL extends the language to make it interactive. LEVEL II COBOL offers many facilities for automatic formatting of a screen and facilitates keying of input.

You can specify areas of the screen into which you are able to key data, and also whether such data is numeric or alphanumeric. You do this by defining the screen as a record in the DATA DIVISION in which the data fields correspond to the input area and FILLER's correspond to the rest of the screen.

An ACCEPT statement makes use of a data item description to input the character positions corresponding to variables with elementary data-names. Conversely, a DISPLAY statement outputs only from non-FILLER fields in the record description which it uses. You can thus easily build up complex conversations for data entry and transaction processing.

While data is being keyed, you have full cursor manipulation facilities,each variable acting as a tab stop. Non-numeric digits may not be entered into fields defined as numeric. Finally, when you have checked that the data is correct, press the RETURN key and the data becomes available to the program. Because all characters are transferred to the appropriate area as they are keyed in, there is no transmission delay.

The following facilities are available for screen layout and formatting:

1. Screen as a record description
2. FILLER
3. REDEFINES
4. AT line column
5. CURSOR addressing
6. Character highlighting
7. Clear screen
8. Numeric validation of PIC 9(n) fields
9. Automatic editing of numeric edited data-items
10. De-editing of numeric edited to numeric data-items

CURSOR CONTROL FACILITIES

During execution of ACCEPT statements the cursor is manipulated on the CRT screen by the cursor control keys on the console keyboard as shown in Table 5-1.

Table 5-1. Cursor and Function Control Keys

Function	ADM-3A	Hazeltine 1520	DEC VT-100
Begin first field	^SHIFT/HOME	HOME	PF1
Begin next field	^J	D arrow	D arrow
Begin previous field	^K	U arrow	U arrow
Forward Space	^L	R arrow	R arrow
Backward Space	^H	L arrow	L arrow
Column Tab	^I	TAB	TAB
Left Zero Fill	.	.	.
Data Entry Terminator	RETURN	RETURN	RETURN

Notes:

1. Where ^ is specified, you must press the "CTRL" (control) key, hold it down and simultaneously press the character key. Back one space for ADM-3A is thus both the "CTRL" and the H character keys.

2. DECIMAL-POINT IS COMMA may be specified in the user program, in which case, the "," character is used for Left Zero Fill.

3. The arrow keys are: D-down, U-up, R-right and L-left.

DISPLAYING DATA ON THE SCREEN

The first step in making your LEVEL II COBOL program interactive is to decide what messages and prompts you want to be displayed on the screen to guide the operator and what action you want the operator to take at each point. This section describes the display facilities. Please note that, as most terminals scroll upwards as a result of a character appearing in the final character position (i.e. bottom right of the screen) it is not possible to use this character position as part of a DISPLAY.

CLEARING THE SCREEN

Unless you are deliberately displaying something upon a screen which you have already displayed, and you know what the result will be, it is advisable to clear the screen before any display. The statement:

```
    DISPLAY SPACE.
or
    DISPLAY SPACES.
```

causes the entire screen to be cleared.

DISPLAYING SINGLE ITEMS

Single text strings such as single prompts or messages can be displayed very easily by using the AT clause to specify the coordinates of the start of the display item on the screen. For example:

```
    DISPLAY data-item-1 AT data-item-2
```

where data-item-2 is PIC 9999, the most significant two digits specify a line number in the range 01 to the maximum number of lines on the screen. The least significant two digits specify a column number in the range 01 to the maximum number of characters per line on the screen. Both numbers are in decimal.

Data-item-1 is the text to be displayed. For example, the following code causes the message SELECT ONE OF THE FOLLOWING ITEMS to be displayed on line 5 of the screen, beginning in character position 5:

```
ENVIRONMENT DIVISION.
      SPECIAL-NAMES.
      CONSOLE IS CRT.
      DATA DIVISION.
      WORKING-STORAGE SECTION.
      01  DISPLAY-ITEM-1          PIC X(33)
          VALUE "SELECT ONE OF THE FOLLOWING ITEMS".
              .
              .
              .
```

```
PROCEDURE DIVISION.
START-OF-PROGRAM.
     DISPLAY SPACES.
     DISPLAY DISPLAY-ITEM-1 AT 0505.
```

Using the DISPLAY...AT... statement, a screen full of information could be built up, one item at a time.

DISPLAYING MORE COMPLEX SCREENS

When several items are to be displayed, many DISPLAY...AT... statements may be required. This can be simplified by declaring FILLER items to fill the intervening gaps, thus requiring only one DISPLAY statement.

For example, to generate:

```
SELECT ONE OF THE FOLLOWING ITEMS

1.  FOOTBALL SCORES
2.  TENNIS RESULTS
3.  GOLF NEWS
4.  EXIT
```

the following program could be used (for an 80 column screen).

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
CONSOLE IS CRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DISPLAY-ITEM-1.
   03 DISPLAY-ITEM-1-1    PIC X(33)
      VALUE "SELECT ONE OF THE FOLLOWING ITEMS".
   03 FILLER PIC X(128).
   03 DISPLAY-ITEM-1-2    PIC X(18)
      VALUE "1. FOOTBALL SCORES".
   03 FILLER PIC X(62).
   03 DISPLAY-ITEM-1-3    PIC X(17)
      VALUE "2. TENNIS RESULTS".
   03 FILLER PIC X(63).
   03 DISPLAY-ITEM-1-4    PIC X(12)
      VALUE "3. GOLF NEWS".
   03 FILLER PIC X(68).
   03 DISPLAY-ITEM-1-5    PIC X(7)
      VALUE "4. EXIT".
      .
      .
      .
```

```
PROCEDURE DIVISION.
START-OF-PROGRAM.
     DISPLAY SPACES.
     DISPLAY DISPLAY-ITEM-1 AT 0505.
```

FILLER items are never actually displayed, not even as spaces, so whatever was on the screen before a DISPLAY will still be displayed in the places covered by FILLER items.

DISPLAYING HIGHLIGHTED TEXT

If the CRT-UNDER clause is used in a DISPLAY statement, the data item is displayed highlighted (provided that your CRT supports this facility, which may be underlining or reverse video characters).

ACCEPTING DATA ENTERED AT A CRT

After you set up and display a screen and prompt the operator to enter some data, the data must be ACCEPTED. Please note that, as most terminals scroll upwards as a result of a character appearing in the bottom right character position, it is not possible to use this character position as part of an ACCEPT.

There are two types of items that may be accepted: elementary data-items and group-items.

ACCEPTING AN ELEMENTARY ITEM

The statement:

```
ACCEPT MYDATA
```

places the cursor at the HOME position and accepts the character string keyed in by the operator until this is terminated by RETURN. This string is directly transferred into the data-item MYDATA, left aligned if too short. MYDATA is then checked against its declaration in the DATA DIVISION and any format errors are reported.

If the AT clause is used, the value of the data-item in the AT clause defines the start position of the ACCEPT data item. This data-item must be PIC 9999, where the most significant two digits define a line number in the range 01 to the maximum number of lines on the screen, and the least significant two digits define a column number in the range 01 to the maximum number of characters per line on the screen. If the data item contains zero or spaces, it is treated as 0101 or, in other words, HOME. The cursor will be positioned at the start of the data item to be accepted, i.e. the position defined by the AT clause (but see Explicit Cursor Positioning).

For example:

```
ACCEPT MYDATA AT 1021
```

positions the cursor at character position 21 on line 10 and accepts whatever is entered there by the operator.

ACCEPTING A GROUP ITEM

Accepting a group item is a little more complex. The group item must be declared in the WORKING-STORAGE SECTION of your program; it will probably bear some resemblance to the data declaration used to generate the DISPLAY screen, except that data items in one will probably be filler fields in the other. It may, in this case, be worth redefining the original DISPLAY group item as the ACCEPT group item. For example:

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
CONSOLE IS CRT.
DATA DIVISION.
WORKING STORAGE SECTION.
01 DISPLAY-ITEM-1
   03 FILLER PIC X(324).
   03 DISPLAY-ITEM-1-1   PIC X(33)
      VALUE "SELECT ONE OF THE FOLLOWING ITEMS".
   03 FILLER PIC X(128).
   03 DISPLAY-ITEM-1-2   PIC X(18)
      VALUE "1. FOOTBALL SCORES".
   03 FILLER PIC X(62).
   03 DISPLAY-ITEM-1-3   PIC X(17)
      VALUE "2. TENNIS RESULTS".
   03 FILLER PIC X(63).
   03 DISPLAY-ITEM-1-4   PIC X(12)
      VALUE "3. GOLF NEWS".
   03 FILLER PIC X(68).
   03 DISPLAY-ITEM-1-5   PIC X(7)
      VALUE "4. EXIT".
01 ACCEPT-ITEM-1   REDEFINES DISPLAY-ITEM-1.
   03 FILLER             PIC X(504).
   03 ACCEPT-ITEM-1-1    PIC X.
   03 FILLER             PIC X(79).
   03 ACCEPT-ITEM-1-2    PIC X.
   03 FILLER             PIC X(79).
   03 ACCEPT-ITEM-1-3    PIC X.
   03 FILLER             PIC X(79).
   03 ACCEPT-ITEM-1-4    PIC X.
PROCEDURE DIVISION.
START-OF-PROGRAM.
     DISPLAY SPACES.
     DISPLAY DISPLAY-ITEM-1.
     DISPLAY ACCEPT-ITEM-1.
```

In the same manner as DISPLAY...AT... the AT clause may be used to define the initial position of the data, thus avoiding an initial FILLER item in the data declaration. The default position for AT is HOME. HOME will also be used if the position defined by AT is outside the physical bounds of the screen.

CURSOR BEHAVIOR DURING AN ACCEPT

Unless explicitly postioned by the program, the cursor will initially be placed at the start of the first data-item to be accepted. While the operator is entering data in response to an ACCEPT, the cursor will advance character by character. If data is entered that does not completely fill the data item, the operator must advance the cursor to the next data item either by advancing one space at a time to the end of the current data item or by using the advance-one-field key. The cursor will not move into FILLER items. At the end of the last data item of a group the cursor will remain in the last character position and the bell will be sounded when any character is typed. The last character typed is the one that will be accepted. Data entry to a group item is terminated by RETURN.

When designing an interactive LEVEL II COBOL program, you should adopt a consistent approach to ACCEPT statements. A number of individual ACCEPTS on the same screen will require the operator to press RETURN at the end of each one, a group ACCEPT performing the same function will require the operator to tab forward from field to field (if the fields are not completely filled by the data entered) and only press RETURN at the end of the last field. A mixture of these approaches in any one program or suite of programs would be confusing for an operator, and should therefore be avoided.

EXPLICIT CURSOR POSITIONING

You can exercise explicit control over the cursor by using the "CURSOR IS data-name" clause in the SPECIAL NAMES paragraph. The data-name must be a PIC 9999 item, where the most significant two digits define a line number in the range 01 to the maximum number of lines on the screen, and the least significant two digits define a column number in the range 01 to the maximum number of characters per line on the screen. On executing an ACCEPT statement, the cursor is moved to the character position defined by the CURSOR data item (if the CURSOR data-item contains zero or spaces or is undefined, HOME is used by default). Any AT clause in the ACCEPT statement still defines the position of the data items on the screen; the CURSOR data-item merely positions the cursor. If the defined position is either outside the physical bounds of the screen or outside the limits of the group item or elementary data item being ACCEPTED, the position is ignored and the start of the first data item is used instead.

If the defined position is in a FILLER item, the cursor moves to the beginning of the next data item. If there is no further data item, the cursor returns to the beginning of the first data item on the screen.

On return from an ACCEPT statement the CURSOR data item contains the address of the final position of the cursor on the screen.

One example of the use of this facility is that in menu-type operations the operator need only move the cursor to a position on the screen corresponding to the selection required. The operator's choice can then be determined by the returned value of the CURSOR data item.

If, in this type of operation, there is one choice per line, the resulting line number could be used for a DEPENDING ON clause and the default choice could be determined by explicitly positioning the cursor on one of the choices before the ACCEPT statement.

Note that in order to use the CURSOR data-item for cursor positioning the data item must contain a value other than zero or spaces. If the CURSOR data item contains zero or spaces, it will not be updated with cursor positions after ACCEPT statements.

The following program could be used to display the sports screen shown earlier and to call a subroutine depending on the response.

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
CURSOR IS CURSOR-POSITION.
CONSOLE IS CRT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CURSOR-POSITION.                                        } CURSOR data items
   03 CURSOR-LINE       PIC 99.                           }
   03 CURSOR-COLUMN     PIC 99.                           }
01 DISPLAY-ITEM-1.
   03 FILLER            PIC X(324).                       }
   03 DISPLAY-ITEM-1-1   PIC X(33)                        }
      VALUE "SELECT ONE OF THE FOLLOWING ITEMS".          }
   03 FILLER PIC X(128).                                  }
   03 DISPLAY-ITEM-1-2   PIC X(18)                        }
      VALUE "1. FOOTBALL SCORES".                         }
   03 FILLER PIC X(62).                                   }
   03 DISPLAY-ITEM-1-3   PIC X(17)                        } DISPLAY item
      VALUE "2. TENNIS RESULTS".                          }
   03 FILLER PIC X(63).                                   }
   03 DISPLAY-ITEM-1-4   PIC X(12)                        }
      VALUE "3. GOLF NEWS".                               }
   03 FILLER PIC X(68).                                   }
   03 DISPLAY-ITEM-1-5   PIC X(7)                         }
      VALUE "4. EXIT".                                    }
   03 DISPLAY-ITEM-1-6   PIC X(32)                        }
      VALUE "POSITION CURSOR AND PRESS RETURN".           }
01 ACCEPT-ITEM-1   REDEFINES DISPLAY-ITEM-1.               }
   03 FILLER            PIC X(504).                       }
   03 ACCEPT-ITEM-1-1    PIC X.                           }
   03 FILLER            PIC X(79).                        }
   03 ACCEPT-ITEM-1-2    PIC X.                           } ACCEPT item
   03 FILLER            PIC X(79).                        }
   03 ACCEPT-ITEM-1-3    PIC X.                           }
   03 FILLER            PIC X(79).                        }
   03 ACCEPT-ITEM-1-4    PIC X.                           }
PROCEDURE DIVISION.
START-OF-PROGRAM.
     DISPLAY SPACES.
     DISPLAY DISPLAY-ITEM-1.
     MOVE 0625 TO CURSOR-POSITION.
     ACCEPT ACCEPT-ITEM-1.
     SUBTRACT 6 FROM CURSOR-LINE.
     GO TO FOOTBALL-SCORES, TENNIS-RESULTS, GOLF-NEWS,
     FINISH-OFF DEPENDING ON CURSOR-LINE.
```

CHAPTER 6

INTERACTIVE DEBUGGING

INTRODUCTION

Two levels of debugging are available to the programmer. Full ANSI debug is available, including optional "debugging lines" that are included if the DEBUGGING MODE switch is present in the SOURCE-COMPUTER sentence. The second is the interactive debug package that is included at run time under the control of the user (see Chapter 2, COMMAND LINE SYNTAX). In certain implementations the ANIMATOR debugging package may also be available.

Note that interactive debug will only work for intermediate code programs; it cannot be used in conjunction with generated code programs.

Note that the interactive debug package reads its commands from the standard input (the ":CI:" device). Consequently, interactive debug may be confusing with programs that read from the standard input (":CI:" reads or ACCEPT's); this includes programs that access program parameters from the command line.

When debug is invoked, it will announce its presence as follows:

```
cbrun +D stock1<<          -command line
COBOL Debug Mark 1.0       -title response
?                          -prompt
```

You now have the following debug commands available, for all of which the lower case letter is accepted:

P - Displays the current program counter (pc)

G - Breakpoint at specified address

X - Execute one LEVEL II COBOL statement at a time

D - Display 16 bytes of a specific address in the DATA DIVISION

D, - Display the next 16 bytes in the DATA DIVISION

A - Replace contents of a memory location by a hexadecimal value or ASCII character

S - Set start of block for correction or display

/ - Display bytes in block above

. - Change bytes in block above

T - Trace paragraphs up to breakpoint specified

Q - Quit interactive debug.

B - Execute statements until the value of a specified memory location changes.

E - Execute statements until the value of a specified memory location changes to a specified value

L - Output one Carriage Return/Line Feed

M - Define debug command macro with name specified

$ - End macro definition

C - Displays specified character

; - Precedes comment to describe a macro just entered

A description of the use of each of these debug commands follows.

THE P COMMAND

The P command displays the address at which the program counter currently points, i.e. where the current instruction is in the PROCEDURE DIVISION code of a program. This hexadecimal address is that printed in the right hand column of a program listing. Note that by default the compiler does not produce the hexadecimal address in the listing. The addressing is produced using the REF directive (See Chapter 2).

EXAMPLE:

At the start of a program the pc is at 0000 as shown below:

```
?P<<                                -command
 0000                               -current pc
?                                   -prompt
```

NOTE:
The location given by the 'P' command is relative to the start of the PROCEDURE DIVISION. All numbers in the debug package are expressed as hexadecimal values.

THE G COMMAND

The G command executes from the current pc until the pc reaches the value in the parameter to 'G'. If this value is not the address of an executed instruction, the breakpoint is never reached and the program continues.

EXAMPLE:

If a breakpoint is required at PARA-22 in the following code:

```
    .                                .
PARA-22.                             017A
    ADD 1 TO COUNT.                  017B  -  hex address
    MOVE FIELD-1 TO FIELD-2.         018C
    .                                .
```

the following command is typed:

```
?G 017A<<
?
```

The display of the second question mark above indicates that the G command has executed completely and thus the breakpoint has been reached.

NOTE: Exactly four hexadecimal digits must be keyed for an address value.

A check on the current address at this point by use of the P command would be as follows:

```
?P<<
 017A                                -returns pc
```

THE X COMMAND

When a suspected error is reached, single instructions can be stepped through one at a time by use of the X command. After each COBOL instruction is executed, the hexadecimal number printed is the address of the first statement on a line. Where COBOL operations are made up of several individual primitive instructions, DEBUG may appear to halt in the middle of a line. If this occurs, press RETURN again.

EXAMPLE:

To single step over the MOVE instruction in the example above the X command sequence would be used as follows:

```
?X<<
 018C
```

THE D COMMAND

To display bytes in the DATA DIVISION, the 'D' command can be used. This displays 16 bytes from the address specified (again the address is derived from the information on the listing). It displays each byte as a hexadecimal value plus an ASCII equivalent if it is printable.

EXAMPLE:

Given the following COBOL data division segment:

```
                .                                    .
02      FIELD-1 PIC XXX VALUE "ABC".          0030
02      FIELD-2 PIC XXX VALUE "XYZ".          0033
02      FIELD-3 PIC X(80) VALUE SPACE.        0036
                .                                    .
```

The D command would be used as follows:

```
?D 0030<<
41-A 42-B 43-C 41-A 42-B 43-C 20-  20-  20- ........
```

After a D command has been used, the command

```
?D,
```

may be used repeatedly to display the next 16 bytes.

THE A COMMAND

The A command is used to amend data at a specified memory location. This correction facility allows continued running even if a bug has produced an erroneous result.

EXAMPLE:

This command may be used to replace the first character "A" of FIELD-1 with the character "G". The value supplied may be a two character hex value or an ASCII character preceded by a double quote, e.g. 47 or "G.

```
?A 0030 47<<                          -amend byte
?D 0030<<
 47-G 42-B 43-C 41-A 42-B 43-C 20-  20-  20-  ........
?
```

THE S COMMAND

Where a number of corrections are required, DEBUG allows specification of a working register that contains an address. This address can be set or incremented and the contents can be displayed or modified immediately by use of the S command. The address and contents can then be displayed by keying '/'.

EXAMPLE:

To display the first byte of FIELD-1 the operation would be as follows:

```
?S 0030<<                                   -load address
?/<<                                        -display
 0030 47-G
?
```

THE '.' COMMAND

To amend the byte at the current location '.' is used; this also increments the working register.

EXAMPLE:

To change FIELD1 to "DEF" the display would be:

```
?S 0030<<                                   -load address
?.44.45.46<<                                -modify
?D 0030<<
 44-D 45-E 46-F ......
?
```

To increment the working register without amending the byte use ','.

THE T COMMAND

An advanced form of the G command is the T command. This also executes up to a breakpoint in the PROCEDURE DIVISION, but also prints the address of each paragraph encountered.

EXAMPLE:

```
?T 017B<<                                   -trace up to 017B
```

THE Q COMMAND

The Q command causes exit from execution of debug and the COBOL program.

THE B COMMAND

The B command allows execution from the current pc until the value of a specified memory location changes.

EXAMPLE:

Assume the following pieces of COBOL code in the DATA DIVISION and PROCEDURE DIVISION respectively:

```
                .                                .
     02         FIELD-1 PIC XXX VALUE "ABC".     0030
     02         FIELD-2 PIC XXX VALUE "XYZ".     0033
     02         FIELD-3 PIC X(80) VALUE SPACE.   0036
                .                                .
                .                                .
 PARA-22.                                        017A  ( hex address )
     ADD 1 TO COUNT.                             017B
     MOVE FIELD-1 TO FIELD-2.                    018C
                .                                .
                .                                .
```

Then the B command would be used as follows:

```
?B 0033<<                    - detect first move to FIELD-2
018C  41-A                   - display of pc and value moved
?
```

THE E COMMAND

The E command allows halting a program when a specified memory location takes on a specified value. In the example above:

```
?E 0033 "A<<                 - detect "A" move to FIELD-2
018C  41-A                   - display of pc and value moved
?
```

THE M COMMAND

You will find that some debug command sequences are used often when debugging. If these sequences are long, it can become tiresome typing them in. To overcome this and to allow the development of complex debugging sequences debug permits the definition of macros comprised both of basic operations and other macros. Macros are given names of one character.

Macros are introduced by the M command followed immediately by the macro name.

EXAMPLE:

To define a macro to execute up to 018C, display the value at 0030, then single-step and display again, the following would be typed:

```
?MZ G 018C   D0030 L X D 0030 $<<
?
```

To invoke this macro its name is typed as follows:

```
?Z<<
41-A 42-B 43-C 58-X 59-Y 5A-Z ...    - first display
0190
41-A 42-B 43-C 41-A 42-B 43-C ...    - second display
?
```

There are two other commands introduced in this macro: L and $.

THE L COMMAND

The L command merely forces a carriage return and line feed to be output on the console.

THE $ COMMAND

The $ command ends a macro definition.

THE C COMMAND

To allow macro writers to output characters to the console, the command 'C' is provided. This outputs its parameter on the console.

EXAMPLE:

```
?C "A<<
A
?
```

THE ; COMMAND

To improve readability of macros, comments may be inserted. These are introduced by the character ';' and terminated by carriage return.

EXAMPLE:

```
?MZ D 0030 X L D 0030 $ ; Run macro<<
```

Macro names must be letters only. Lower case letters are converted internally to upper case.

If an error is made in typing in a macro, it may be re-entered. However, there is only a finite amount of macro space, and space is not released if a macro is re-entered. If the space runs out or the maximum nesting of macros is exceeded, the message STACK OVERFLOW will appear. After this overflow crash has occurred, the debug system will return to command mode and will generally tidy up the stack to allow you to continue. However, if more serious crashes occur, i.e. those with no message, then the system will not recover.

For a convenient reference to debug commands see Appendix G.

CHAPTER 7

MULTI-LANGUAGE CALL FACILITIES

INTRODUCTION

LEVEL II COBOL enables segmented COBOL programs to be called from a main application program, and also enables programs written in other languages to be called from a main COBOL application program.

CALLING COBOL PROGRAMS

An application written in COBOL may be arranged into a number of separate COBOL programs, which communicate and invoke each other by use of the COBOL CALL verb.

FORMAT OF LEVEL II COBOL CALL

The general format of the LEVEL II COBOL CALL verb is given in the LEVEL II COBOL Language Reference Manual.

FORM OF LEVEL II COBOL PROGRAMS

Each program in a LEVEL II COBOL application suite must be written in COBOL and with the exception of the main program should have a LINKAGE SECTION in the DATA DIVISION with which to communicate with other COBOL programs.

A COBOL program that is CALLed by the main program of a segmented application may be an intermediate code program or a generated code program; that is, the two types may be freely mixed in an application. However, the intermediate or generated code file must be accessible at run time.

RUN-TIME PROGRAM LINKAGE

Run-time execution of the COBOL verb CALL depends on the parameter used by the call. COBOL programs and C language subroutines, or subroutines in other languages, can be called.

When the parameter is an alphanumeric quantity its value is interpreted as a filename and the appropriate file of intermediate or generated code is loaded into memory and executed. If the filename includes an extension, the appropriate intermediate (.INT) or generated (.GNT) code file is located. However, if there is no extension, the Run-Time System (RTS) uses the following convention:

1. It tries to find the specified file.

2. If this fails, it appends .GNT to the filename and tries to find the resulting generated code file.

3. If this fails, the RTS appends .INT to the filename and tries to find the resulting intermediate code file.

When the parameter is a numeric quantity, its value is interpreted as a call number to a run-time subroutine table, and the corresponding machine code subroutine is executed. However, the subroutine must first have been incorporated into the RTS (See CALLING RUN-TIME SUBROUTINES, below).

EXAMPLE LINKAGE

```
PROCEDURE DIVISION.
        .
        .
CALL "subitm.INT" USING ...
        .
        .
CALL "10" USING ...
        .
        .
```

For the first CALL in this example to perform correctly the file subitm.INT must be present in the current directory and must contain a compiled COBOL program. For the second CALL to perform correctly the RTS must contain a machine code subroutine arranged as subroutine 10.

SAMPLE APPLICATION - USER INTER-PROGRAM COMMUNICATION

Figure 7-1 shows a sample CALL "tree".

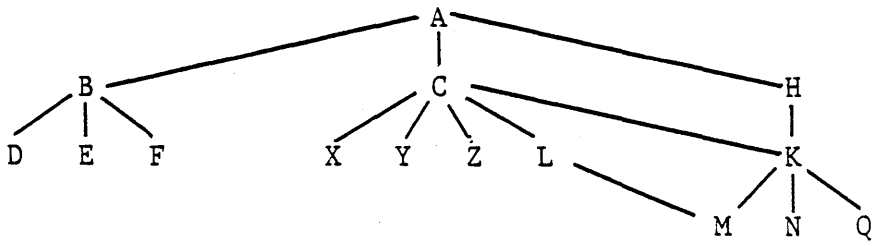


Figure 7-1. Sample CALL Tree Structure.

The main program A, which is permanently resident in memory, calls B, C, or H, which are subsidiary and stand-alone functions within the application. These programs call other specific functions as follows:

B calls D, E and F
C calls X, Y or Z conditionally and K or L conditionally
H calls K
K calls M, N or Q conditionally
L calls M if it needs to.

As the function B, C and H are stand-alone, they do not need to be permanently resident in memory together, and can therefore be called as necessary using the same physical memory when they are called. The same applies to the lower functions at their level in the tree structure.

In the example shown in Figure 7-1, the use of CALL and CANCEL would need to be planned so that a frequently called subroutine such as K would be kept in memory to save load time. On the other hand, because it is called by C or H it cannot be initially called without C or H in memory, i.e., the largest of C or H should call K initially so as to allow space. It is important also to avoid overflow of programs. At the level of X, Y and Z it does not matter in which order loading takes place, because they do not make calls at a lower level.

Called programs that open other files should be left in memory. This avoids them having to re-open files on every call. EXIT does not close files but CANCEL does.

THE CANCEL STATEMENT

The CANCEL statement releases memory occupied by the cancelled program and closes any files opened by it. If a tree structure of called independent programs is used as shown above, each program can call the next dynamically by using the technique shown in the following sample coding:

```
WORKING STORAGE SECTION.
01   NEXT-PROG          PIC X(20) VALUE SPACES.
01   CURRENT-PROG       PIC X(20) VALUE "FSTPRG.INT".
PROCEDURE DIVISION.
LOOP.
     CALL CURRENT-PROG USING NEXT-PROG.
     CANCEL CURRENT-PROG.
     IF NEXT-PROG = SPACES STOP RUN.
     MOVE NEXT-PROG TO CURRENT-PROG.
     MOVE SPACES TO NEXT-PROG.
     GO TO LOOP.
```

The actual programs to be run can then specify their successors as follows:

```
.
.
.
LINKAGE-SECTION.
01   NEXT-PROG          PIC X(20).
.
.
.
PROCEDURE DIVISION USING NEXT-PROG.
.
.
.
.
.
MOVE "FOLLOW.INT" TO NEXT-PROG.
EXIT PROGRAM.
```

You can see that in this way, each independent segment or sub-program cancels itself and changes the name in the CALL statement to call the next one by use of the USING phrase.

LIMITATIONS OF CALL

Any number of COBOL programs and C subroutines can be CALLed from a COBOL program. Operational limitations on CALL are as follows:

1. The CALLed intermediate or generated code program file must be present at the time of the first CALL to the file.

2. There must be room available in memory for the program to be loaded.

3. Run-time subroutines must be pre-configured into the RTS.

The memory constraints are aided by the following:

1. The ON OVERFLOW verb detects whether a CALL has failed due to lack of available memory space.

2. The CANCEL verb reclaims unused storage when executed at run time.

With these facilities available, large and complex LEVEL II COBOL application program suites can be run. System designers in particular should realize that the total size of the application is not constrained by the intrinsic hardware environment.

CALLING RUN-TIME SUBROUTINES

The RTS has been designed to allow the incorporation of user-written subroutines within the run-time system itself; these can then be accessed by user programs utilizing the COBOL CALL verb with a numeric argument. In this way, functions for which COBOL is not appropriate, (such as low-level data conversion), may be written in C, or some other language.

The issued RTS contains no user routines, and so an attempted CALL with a numeric argument will result in an RTS error. The Installation Guide describes the steps necessary to incorporate user routines into your RTS.

IMPLEMENTATION

When the RTS processes the intermediate code for a CALL, it goes through an internal CALL executor, which determines the type of call and accesses the appropriate table to find the routine. All of the arguments following USING are COBOL data names; the absolute addresses of these items are placed in the global calargv[] array, and the count is assigned to calargc. These are analogous to the argv/argc structure with which a C program is invoked. For instance, if a COBOL program included the following sentence:

```
CALL "1" USING A, B.
```

then calargv[0] would contain the address of data item A, calargv[1] would contain the address of data item B, and calargc would have the value 2. The data items may be numeric or alphanumeric strings. Note that if COMP variables are used, attention must be paid to the fact that the byte order may differ from that of the machine. In COBOL, the highest order byte is always first.

If the CALL parameter is numeric, as in the example above, this indicates that the call is to a user routine in the RTS, rather than to another COBOL program, and the xequcall() function is called. This function contains only a single "switch" statement; each "case" in this statement is a call to a user subroutine.

CHAPTER 8

LEVEL II COBOL APPLICATION DESIGN CONSIDERATIONS

Anyone designing a COBOL application program requires it to make efficient use of the space and facilities available. This chapter is written for someone designing an application to be written in LEVEL II COBOL and describes:

* The factors that you must consider when deciding whether or not to produce a generated code version of your programs.

* The technique of segmentation (dividing large programs into smaller units).

* Some ways of producing more compact and efficient code.

INTERMEDIATE OR GENERATED CODE?

A generated code program will usually execute more quickly than an intermediate code program. The Run-Time System (RTS), recognising a generated code file, can execute each generated code instruction directly on the host processor rather than having to interpret it.

However, there is a qualification to this. Input-output operations are still handled interpretively by the RTS. Consequently, a program that is "I-O bound" (that is, spends most of its time moving data to and from files and devices rather than performing arithmetic on it) will derive relatively little benefit in speed from code generation. Only programs that are "processor bound" (that is, spend most of their time operating on data rather than transferring it) are likely to increase their run-time speed significantly as a result of code generation.

Another consideration is space. Since intermediate code is very compact, the generated code version of a program will take up more space than its intermediate code equivalent. In fact, the only difference between an intermediate and generated code file lies in the code area of the file; the data areas are identical. Typically, the code area of a generated code program will be a little less than twice the size of the code area of the equivalent intermediate code program.

In summary, code generation is worth while if:

* Your program is "processor bound" rather than "I-O bound".

* You have enough memory and disk space available to cope with the space overheads of generated code. You should allow for the code area of an intermediate code file to approximately double in size in the generated code version.

There is also the consideration that you cannot use interactive debug or, if available, ANIMATOR with a generated code program. However, this is less important, since code generation should really only be considered when an application has been thoroughly tested.

If your program is quite large, it is likely that you have designed it in separate segments (see the next section, SEGMENTATION (OVERLAYING)). Since a segmented program may freely mix intermediate and generated code, you have the opportunity to use code generation on just those parts of the application that will most benefit from it. For example, input/output routines could be left as intermediate code, while pure processing routines could be optimised by code generation.

SEGMENTATION (OVERLAYING)

LEVEL II COBOL enables a COBOL program with a large PROCEDURE DIVISION to be divided into a COBOL program with a small PROCEDURE DIVISION and multiple overlays containing the remainder of the PROCEDURE DIVISION. The resident part is known as the permanent segment and the overlays are known as independent segments.

By use of the LEVEL II COBOL segmentation feature, all of the PROCEDURE DIVISION can be loaded into the available memory. However, because it cannot be loaded all at once, it is loaded one segment at a time to achieve the same effect in the reduced memory space, as shown below.

In the case of a COBOL segmented program, the compiler allows space for the largest segment in that program:

```
                        -Segment 1-
---PERMANENT SEGMENT----Segment 2--
                        -Segment 3-------
                        -Segment 4----

<---------maximum 60K bytes------------>
```

This is the maximum of the abstract machine per called program.

It may be less on some machines. For a segmented program, the beginning of each segment in the PROCEDURE DIVISION is denoted in the LEVEL II COBOL source code by a SECTION label. For example:

```
        .
        .
        .
SECTION 52.
     MOVE A TO B.
     etc.
        .
        .
        .
SECTION 62.
     MOVE X TO Y.
     etc.
        .
        .
        .
```

Segmentation can be applied only to the PROCEDURE DIVISION. The IDENTIFICATION, ENVIRONMENT and DATA DIVISIONs are common to all segments; in addition there may be a common PROCEDURE DIVISION. All this common code is known as the permanent segment. Control flow between permanent and independent segments is fully specified in the LEVEL II COBOL Language Reference Manual.

PRODUCING COMPACT AND EFFICIENT CODE

This section offers some advice on how to improve the run-time performance of intermediate and generated code.

OPTIMISING INTERMEDIATE CODE.

Declaring data items to have usage COMP causes them to be stored compactly in the minimum number of bytes needed to accomodate in binary format the largest number allowed by the PICTURE string. Note that an eight-bit byte is assumed by the LEVEL II COBOL system. However, declaring usage COMP does not automatically ensure that arithmetic on such items will be efficient as well as compact. Except for the special cases detailed below, arithmetic on COMP data items is done by expanding them in internal registers to BCD format, and re-converting to COMP when storing the result.

Efficient coding (known as COMP code) is available for the following types of statements, provided the data is in the correct format (the COMP compiler directive need not be specified).

1. {ADD / SUBTRACT} source {TO / FROM} target

where either both source and target are PIC 9(2) COMP, or both are PIC 9(4) COMP or the source is an unsigned integer literal less than 256 and the target is PIC 9(2) COMP, or the source is an unsigned integer literal less than 65536 and the target is PIC 9(4) COMP and there is no ON SIZE ERROR clause.

2. {MULTIPLY / DIVIDE} source {TO / FROM} target

where both source and target are PIC 9(4) COMP and there is no ON SIZE ERROR clause.

In such cases arithmetic is done on one- or two-byte binary quantities without overflow checking and with binary wrap-around in eight or 16 bits.

3. MOVE source TO target

where the source and target satisfy the rules given above for ADD and SUBTRACT.

In this case the MOVE is a one- or two-byte transfer without data conversion.

4. Comparisons of the form

left-operand relation right-operand

where the operands again satisfy the rules given above for ADD and SUBTRACT except that either (not just the left- hand one, but not both) may be a literal.

This yields a raw binary one- or two-byte comparison.

5. Finally, even more compact and efficient code is generated for a statement of the form

IF operand relation literal GO TO label

where the operand is declared as PIC 9(2) COMP and is the first data-item in the WORKING-STORAGE section. The literal is also an unsigned integer less than 256, and there is no ELSE clause. This is known as a "Special IF".

In case 4, the efficient code can be generated even when the comparison is just one of a sequence connected by AND/OR; however, format 5 is totally specific.

Code compiled for these statement formats runs considerably faster than equivalent non-compact code, so it is worth taking care to use these formats where possible. However, we must now examine the interaction between the semantics detailed above and the ANSI COBOL specification. The specific code facilities detailed below will be enabled only if the COMP directive is specified in the command line (See Chapter 2). The following considerations are relevant:

1. If there is an ON SIZE ERROR clause, the target must not be affected if there is numeric overflow; COMP code is never generated in such a case. However, if there is no ON SIZE ERROR clause, the result on numeric overflow is implementor-defined.

   In LEVEL II COBOL using COMP code, the result is defined as above, i.e. binary-byte oriented arithmetic with 8 or 16 bit wrap-around.

   You may decide to take advantage of this extra level of definition as a LEVEL II COBOL extension, although your programs may not then be portable to other ANSI COBOL compilers because you will be depending on a feature that is undefined in ANSI COBOL; alternatively, if you know that your arithmetic statements will not lead to numeric overflow, your programs will be portable in any case.

2. Unsigned subtraction: when the result is negative, ANSI COBOL requires that the absolute value is stored. COMP code stores the two's complement result. Because of this conflict with ANSI COBOL semantics, COMP code is never generated for SUBTRACT statements unless you specify the COMP directive to the compiler; you should do this either when you know your unsigned COMP subtractions will not underflow (in which case your programs compiled with COMP code will remain portable) or when you wish to take advantage of the non-standard behavior on underflow.

3. Truncation on MOVE literal: in the statement:

   MOVE literal TO target

   where the target is PIC 9(2) COMP and 99 < literal < 256 or the target is PIC 9(4) COMP and 9999 < literal < 65536, ANSI COBOL requires that the literal is truncated to the number of decimal places specified for the target. COMP code does not truncate, but stores the binary value. As in case 2 above, because of this conflict the compiler will not generate COMP code for this form of statement unless, for either of the reasons described above, you specify the COMP directive.

If ANIMATOR is to be used, it should be noted that the COMP code generated for the special IF statment involving the first byte of working-storage causes the whole statement, including the GO TO, to appear to ANIMATOR as a single operation, so that in single-stepping the cursor will not stop on the GO TO. Because of this anomaly, COMP code is not generated for this format unless you specify the COMP directive.

OPTIMISING GENERATED CODE

Alphanumerics

COBOL statements such as MOVE and COMPARE on alphanumeric data-items can be improved at run time. In generated code, these statements are converted to string instructions or copy loops.

Comparisons involving a collating sequence are slower, but still run at greater speeds than intermediate code. This is because the collating sequence is checked after every mismatch.

Numerics

Two-Operand MOVE, ADD and SUBTRACT Statements

Two-operand instructions on numeric data-items are especially optimized by code generation. Generally, this means that statements such as:

```
MOVE A TO B.
ADD  A TO B.
SUBTRACT A FROM B.
```

can be highly optimised if the source and target are of the same usage and alignment, and if the data-items are unsigned or have the default sign (trailing included).

These kinds of operation are optimised if the usage is COMP or DISPLAY. COMP is much faster than DISPLAY, and both are faster than the general arithmetic operations described in the next section. These special optimisations do not apply if the usage is COMP-3, although the speed of COMP-3 operations is improved (see the LEVEL II COBOL Language Reference Manual for more information on COMP-3).

NOTES:

1. At code generation, a literal source is altered to match the usage and alignment of the target. This is because MOVE, ADD and SUBTRACT operations on similar data-items are optimised with generated code.

2. MOVE statements involving a similar source and target are particularly fast when generated, because this statement involves no arithmetic.

3. MOVE statements where the USAGE is COMP are less efficient if the data-item has to be truncated.

General Arithmetic Statements

MULTIPLY, DIVIDE, COMPUTE and more complex MOVE, ADD and SUBTRACT statements can have run-time speeds improved by code generation. Statements such as:

```
MOVE A TO B,C.
ADD A,B TO C.
SUBTRACT A FROM B,C.
MULTIPLY A BY B.
DIVIDE A INTO B GIVING C.
COMPUTE A = B+C.
```

all run faster when generated, although these statements are not as fast as two-operand ADD, SUBTRACT and MOVE statements. Generally, where MOVE, ADD and SUBTRACT statements have more than one source, these run slower than simple statements such as ADD A TO B.

For these general arithmetic statements, where the usage is COMP, run-time speeds are slower than usage DISPLAY or COMP-3.

Subscripts and Indexexes

Generation can improve the run-time speeds of COBOL code containing data-items that are subscripted or indexed. Subscripted or indexed data-items whose usage is COMP have very fast run-time speeds with generated code. Similarly, where the usage is DISPLAY or COMP-3, the run-time speed is improved, although not as fast as usage COMP.

COMP Subset and Control Flow

COBOL programs containing code written in COMP subset are optimised by generation. The previous section describes COMP subset in the section on producing compact and efficient code. Similarly, statements controlling the flow of the program (PERFORM, GO TO, IF...THEN, etc.) are optimised by generation. This is because the generator can convert these statements into a very few machine instructions.

Tips for Writing a Program

You can optimise COBOL programs by following these guidelines when writing a program:

1. Use two-operand arithmetic as much as possible.

2. Use source and target data-items of the same usage and alignment.

3. Select USAGE DISPLAY and USAGE COMP carefully to greatly increase the speed of execution. COMP data-items execute at the fastest speeds if two-operand arithmetic statements are used; DISPLAY is fastest when the more complex arithmetic statements are used.

4. For the highest performance, use the COMP subset. To achieve maximum speeds, the COBOL program must be carefully written to use only operations within the subset. Generally, the COMP subset is not suitable for general applications programs.

CHAPTER 9

FILE AND RECORD LOCKING FOR INDEXED FILES

The file and record locking features discussed here are based on the IBM 8100 DPPX multi-user specification, where all the file sharing attributes are specified by the user through COBOL language syntax and appropriate semantics. This release implements selected locking capabilities according to the above specification.

New and existing LEVEL II COBOL programs must be compiled with the FILESHARE directive (See Chapter 2) to enable extended locking syntax. Also, you must modify file error status actions to take advantage of file and record locking features (See ERROR STATUS later in this chapter).

There are two types of file locking: kernel locking and creation locking.

KERNEL LOCKING

If your UNIX system provides kernel locking, the UNIX kernel will provide the necessary internal mechanisms for handling the file and record locks used by LEVEL II COBOL programs.

CREATION LOCKING

Creation locking is a method by which files and records can be locked in systems that do not provide some form of kernel locking. The mechanism involves the creation of two disk files, one in the current directory and one in the /isam (for Indexed Sequential Access Method) directory.

The file created in the current directory has the ".lok" suffix attached to the basename of the data file (e.g. basename.lok). It contains a lock table that indicates individual file and record locks held by the run units accessing the file. The ".lok" file is always present, but the values contained there change as locks are acquired and released. This file should not be removed, as all processes that access the data file also use the same lock file.

The file created in the /isam directory contains lock files that are essentially semaphores indicating that a particular indexed sequential file is being accessed. In this way, no two processes have access to a "busy" indexed sequential file at the same time. The lock is acquired by a run unit automatically upon entry to the ISAM package in the Run-Time System (RTS) and is released automatically upon exit. However, care should be taken to manually remove the appropriate lockfile in the /isam directory if one remains due to premature program termination. The name of the lockfile indicates the process that acquired it.

ISAM FILE LOCKING FUNCTIONS

A lock is a mechanism that regulates concurrent access to a file or record. This allows integrity to be maintained in an environment in which concurrent run units may access the same file. A run unit is a set of one or more object programs that function as a unit within one user environment at run time. Locking protects a file or record in use from the updating operations of concurrent run units, i.e. other user programs. A file lock is associated with a file, and a record lock is associated with a record.

A run unit holds a file lock for each file that is open to the run unit. The lock mode and type of lock for timeshared files are determined by:

1. A statement in the file control entry, as described in this chapter.

2. Compile-time parameters specified as part of the compiler command line.

FILE LOCK MODES

File lock modes are specified in the SELECT clause of the FILE-CONTROL paragraph.

EXCLUSIVE RECORD LOCKING

When a run unit holds an exclusive lock, concurrent run units are prevented from opening the file that is locked. This also prevents individual record locking from occurring within the file. A non-exclusive file lock allows concurrent run units to open the file associated with this run unit. Record locking can therefore occur within this file.

AUTOMATIC RECORD LOCKING

A record is automatically locked whenever it is accessed by a run unit.

MANUAL RECORD LOCKING

A record is locked by an access only if the statement causing the access specifically locks the record.

TYPES OF FILE LOCK

There are three types of file lock that can be specified by the user in the compiler command line:

1. Restricted. A restricted file is one that can be exclusively locked to one run unit.

2. Uncommittable. An uncommittable file is one that can be shared with a record locked to one unit. The lock is released upon the next access of that run unit to the file containing the record.

3. Committable. A committable file is one that can have several records locked to one run unit. Locks are not released until the next "quiet point", i.e. after run unit termination or immediately after a commitment. Note that this differs from the DPPX definition of committable.

By default, indexed files are uncommittable. For details of changing the defaults by the cobol command, see Chapter 2.

INDEXED FILE SPECIFICATION

The default lock mode is AUTOMATIC for files with ORGANIZATION INDEXED, whether in I-O, INPUT or OUTPUT mode. A record lock is thus acquired by the execution of the READ and START statements referencing the file. It is only released on next access to the file, i.e. at the end of execution of the next I-O statement that references the file.

The COBOL syntax for locking of indexed files is specified in the ENVIRONMENT and PROCEDURE divisions as shown below.

ENVIRONMENT DIVISION

Extra COBOL syntax is not mandatory to specify locking, as the default is AUTOMATIC. The file control entry format is shown below:

```
FILE-CONTROL.

SELECT file-name

ASSIGN TO {external file name literal}[ {external-file-name-literal} ]
          {file-identifier           }[ {file-identifier           } ]

;ORGANIZATION IS INDEXED

[ACCESS MODE IS    {SEQUENTIAL}]
                   {RANDOM    }
                   {DYNAMIC   }
[

;LOCK MODE IS      {EXCLUSIVE }
                   {AUTOMATIC }
                   {MANUAL    }

;RECORD KEY IS data-name

[ALTERNATE RECORD KEY IS data-name-2 [with DUPLICATES] ...]

[FILE STATUS IS data-name]
```

The full specification of the file control entry is contained in the LEVEL II COBOL Language Reference Manual.

The only part of the file control entry that is specific to locking is the LOCK MODE clause. When this clause is omitted, LOCK MODE IS AUTOMATIC is assumed.

When LOCK MODE IS EXCLUSIVE is specified, an exclusive file lock is acquired by the run unit when the file is opened. While a run unit holds an exclusive file lock, concurrent run units cannot open the associated file, and record locking does not occur within that file.

When LOCK MODE IS AUTOMATIC or LOCK MODE IS MANUAL is specified, a non-exclusive file lock is acquired by the run unit when the file is opened. While a run unit holds a non-exclusive file lock, concurrent run units can open the associated file, and record locking may occur. However, other run units are prevented from opening the file with an exclusive lock.

If LOCK MODE IS AUTOMATIC is specified for an unrestricted file, and the file is open for I-O, a record lock is acquired by the execution of the READ, WRITE, REWRITE and DELETE statements referencing the file.

If LOCK MODE IS MANUAL is specified for an unrestricted file, a record lock is acquired by the execution of a READ statement referencing the file only if the READ statement includes the WITH LOCK phrase. Note that if a file is open, I-O with a record locked any attempt to read the record will result in the RTS error number 68 -- record locked. This is true even if the READ statment does NOT contain the "WITH LOCK" phrase.

Tables 9-1, 9-2, and 9-3 describe when record and file locks are acquired and released.

Table 9-1. Record Lock for Committable Indexed Files.

Statement	Lock Mode	Opened	Lock Acquired
READ WITH KEPT LOCK	AUTOMATIC AUTOMATIC MANUAL MANUAL	I-O INPUT I-O INPUT	Yes No Yes Yes
READ (without WITH KEPT LOCK)	AUTOMATIC AUTOMATIC MANUAL MANUAL	I-O INPUT I-O INPUT	Yes No No No
START, WRITE DELETE, REWRITE	AUTOMATIC/ MANUAL	Any of the OPEN options	Yes

Table 9-2. Record Lock for Uncommittable/Unrestricted Indexed Files.

Statement	Lock Mode	Opened	Lock Acquired
READ WITH LOCK	AUTOMATIC AUTOMATIC MANUAL MANUAL	I-O INPUT I-O INPUT	Yes No Yes No
READ (without WITH LOCK)	AUTOMATIC AUTOMATIC MANUAL MANUAL	I-O INPUT I-O INPUT	Yes No No No
START, DELETE WRITE, REWRITE	AUTOMATIC/ MANUAL	I-O	Yes
START	AUTOMATIC/ MANUAL	INPUT	No
WRITE	AUTOMATIC/ MANUAL	OUTPUT	No

NOTE: A record lock for an uncommittable file is acquired for a START or READ statement and is released on the next I-O request for the file. For a record lock acquired by a WRITE, REWRITE, or DELETE statement, a lock is released at the end of the execution of this I-O statement.

Table 9-3. File Lock for Indexed Files.

File Type	Acquired at	Released
Committable	OPEN	At CLOSE.
Uncommittable	OPEN	At CLOSE.

PROCEDURE DIVISION

There are two statements specific to locking in the PROCEDURE DIVISION: COMMIT and ROLLBACK. Note that this implementation differs from the IBM 8100 DPPX specification of COMMIT and ROLLBACK, in that all file modifications happen immediately.

The COMMIT Statement

The COMMIT statement releases record locks and specifies a quiet point. Execution of the COMMIT statement has no effect in programs containing only uncommittable files.

Format: COMMIT

Execution of the COMMIT statement causes the following action to occur:

1. All record locks in the committable files held by the run unit are released.

2. A new quiet point is established for the run unit.

For each committable file open to the run unit, the position of the current record pointer is undefined.

If the execution of the COMMIT statement is unsuccessful, the run unit is abnormally terminated.

The ROLLBACK statement

Execution of the ROLLBACK statement has no effect. This means that other users do not have access to a copy of the original record after changes have been written to it. Other users do, however, have 'read only' access to the modified record.

Format: ROLLBACK

ERROR STATUS

Programs that intend to update indexed data files and which need to use locking must provide routines for handling any non-fatal errors that may arise. Programs handle such errors by the inclusion of the FILE STATUS IS identifier-name line in the SELECT statement for the appropriate file. In addition, appropriate code must be provided in the PROCEDURE DIVISION to handle specific returned error values. Omission of the FILE STATUS IS phrase can cause non-fatal ISAM errors to be treated as fatal.

When STATUS has been selected and an error occurs, status bytes 1 and 2 together provide a description of the error (See the LEVEL II COBOL Language Reference Manual for more information). When status byte 1 is returned with a "9" due to an ISAM error, status key 2 will contain a letter in the range A through H. The decimal value of the ASCII representation of the returned character corresponds to the run-time system error number. For example, "9D" represents RECORD-LOCKED. See Appendix E for a complete list of run-time errors.

The RTS itself takes no action on non-fatal errors when STATUS has been selected except in the following cases:

1. For imperative statements following INVALID KEY and AT END (when these are provided in I-O statements).

2. For USE... procedures in the DECLARATIVES SECTION (for errors which set status byte 1 to "2". So, if STATUS has been selected, the program must explicitly check the status after every statement where an error might arise. In addition, a default mechanism should be provided for graceful termination of the program in case of unanticipated error conditions.

If at any stage the record pointed to by a currently running program has been deleted by another program, the current program's record pointer will be updated to point to the next record in the file.

If, however, a lock is encountered on attempting to access a record (i.e., a "D" is returned as error status key 2), the current record pointer is undefined except in the case of:

READ NEXT or START ... KEY IS NOT

In these cases, the current record pointer is updated as if the lock had not been encountered and the operation had completed.

FORMS-2 users should note that the ISAM maintenance programs generated by FORMS-2 do not select STATUS bytes, and thus require some additional programming to use record locking. Otherwise, the RECORD-LOCKED condition will be a fatal I-O error resulting in an RTS error message stating that the record was locked, and termination of the program.

FILE LOCK COMPILER CONTROLS

Compilation of user programs that incorporate automatic uncommittable file locking is exactly as for single-user COBOL programs. This default type of lock for files with indexed organization can be altered, however, by use of a directive in the compiler command line.

See Chapter 2 for a full description of these directives.

THE FILESHARE COMPILER DIRECTIVE

In order to use the extended file and record locking syntax described in this manual, a program must be compiled with the FILESHARE directive. See Chapter 2 for information on how to do this.

There are two directives that can change the default lock mode for indexed files:

```
COMMIT"(INDEXED)"
```

which means "treat indexed files as committable", and:

```
RESTRICT"(INDEXED)"
```

which means "treat indexed files as restricted.

THE LOCKDEMO DEMONSTRATION PROGRAM

Lockdemo is a data entry program that creates name and address records in an indexed sequential data file, "customer", together with this file's index, "customer.idx". It specifically tests automatic locking for uncommittable files, which is the default for indexed files.

After a record has been entered or altered on the screen, it is processed by positioning the cursor to the appropriate selection asterisk on the screen and pressing the RETURN key.

The operator has the following selections:

```
Display NEXT Record
FIND specified record
ENTER a new record
UPDATE an existing record
```

These correspond to the COBOL I-O functions:

```
READ NEXT
READ by key
WRITE
REWRITE
```

To select the file process required, and also to move the cursor from one data input field to another, the operator uses LEVEL II COBOL cursor control functions. These are usually the arrow, tab, and home keys. However, the keys that control these functions may be different on different terminals; refer for more information to Chapters 5 and 10.

Using cursor control functions, the operator may "tab" the cursor forward and backward from one data input field to the next. The cursor may also be moved forward and backward in data input fields, non-destructively, one character position at a time.

It may be HOMEd to the first character position in the first data input field. In addition, there is a numeric validation on the telephone field which permits only numeric characters to be entered.

USING LOCKDEMO

To use the record locking demonstration program first compile the program and then type:

```
cbrun lockdemo<<
```

This loads lockdemo for User-1. To demonstrate locking, another user must execute lockdemo concurrently. Type on another terminal:

```
cbrun lockdemo<<
```

which loads lockdemo for User-2.

This clears the screen, then displays the following:

```
LOCKING DEMONSTRATION PROGRAM
```

This program demonstrates the use of record locking facilities using an ISAM file and I-O access mode. You can specify actions as follows:

```
* NEXT record  * FIND record  * ENTER record  * UPDATE record  * EXIT
(Position cursor over appropriate asterisk and press RETURN)

--------------------------------------------------------------------------
         NAME:      [                                  ]
         ADDRESS:   [                                  ]
                    [                                  ]
                    [                                  ]
         TELEPHONE: [            ]
--------------------------------------------------------------------------

         (Last action was       , and it          )
```

To create a record, move the cursor into the data area and key data into the unprotected areas defined by square brackets. When a record is complete, move the cursor to the "ENTER record" asterisk and press RETURN. The record will be written to disk. If it was correctly entered, the unprotected areas will then be space filled and ready for the next record to be entered.

To find a particular record, move the cursor to the NAME field and key the name only followed by the RETURN key. Move the cursor to the asterisk for "FIND record" and press RETURN. The appropriate record will then be displayed.

If you try to FIND a record that does not exist, a message is displayed:

```
INVALID ACTION:  UPDATE NEW OR ENTER EXISTING RECORD
```

If you reach end-of-file, the message:

END OF FILE -- FIND, ENTER OR EXIT OPTIONS

is displayed.

To alter a record once it has been found, change the appropriate field, select the UPDATE record option and press RETURN.

To display the next record at any time, select "NEXT" (to which the cursor returns by default after any action) and press RETURN. If the end of file has been reached, the END OF FILE message will be printed. If "NEXT" is requested again, however, the pointer will return to the beginning of the file and the first record will be displayed.

After any file process, the line beneath the data entry screen:

(Last action was , and it)

is completed with the appropriate action (selection) and either COMPLETED or FAILED, as the case may be.

If you press RETURN with the cursor in the data entry area a message is displayed:

POSITION CURSOR AT ACTION POSITION AND TRY AGAIN

You have not lost the data in the data fields. Move the cursor to the desired selection and press RETURN.

OPERATING LOCKDEMO AS A SECOND USER

Having performed the operations given above as User-1, you will have built a data file and written a few records, and the record currently being accessed will be non-exclusively locked by User-1. Non-exclusive locking enables files to be opened, but individual records are locked if currently being accessed by another user.

Move to the User-2 terminal and ENTER a new record. Note that the file will be opened and the record written to disk. Now try to access (with "FIND" or "NEXT") the record that is currently displayed on the terminal of User-1. The line beneath the data entry area will be displayed as:

(Last action was "selection", and it LOCKED)

where "selection" is "FIND" or "NEXT".

Another message will then be displayed after any disk access failure:

RECORD LOCKED - FIND, ENTER OR EXIT OPTIONS

You can now try again to access the locked record by pressing RETURN. If the record is still locked the error display is repeated.

To terminate the run on any terminal, select the EXIT option and press RETURN.

PROGRAM SOURCE DESCRIPTION

The program listings are reproduced in Appendix K. The features are described in this section and are cross-referenced to the source code. The COBOL data description code for the display screen is copied in from the FORMS-2 generated lockdemo.DDS file at compile time; details of such files can be found in the LEVEL II COBOL FORMS-2 Utility Manual.

The first thing to notice on the listings is the OPTIONS SELECTED line which shows that the option RESEQ was selected at compile time in the compiler command line. This resulted in the line numbers on the left of the listings being generated in sequence.

In this program the IDENTIFICATION DIVISION, which is not mandatory in LEVEL II COBOL, has been used only to name the program.

In the ENVIRONMENT DIVISION the SPECIAL-NAMES paragraph specifies two LEVEL II COBOL extensions to standard COBOL: CONSOLE IS CRT specifies that you wish to use the extension that enables records to be entered as individual screens of data; CURSOR IS CURSOR-POSITION specifies a named area (COBOL data item) that will contain the position of the cursor on the screen at any time during the program execution. The data-item is defined in the DATA DIVISION following the ENVIRONMENT DIVISION.

The next section in the ENVIRONMENT DIVISION specifies input/output handling in the program, which includes access to the terminal and the disk file. The name given to the disk file for internal use in the program is CUSTOMER-FILE and this is assigned to the external operating system file "customer". The organization is specified as INDEXED (sequential) with DYNAMIC ACCESS, allowing records to be accessed sequentially or at random by key. The data-item to contain the key by which records will be found is named CUST-KEY; the data-item to contain the status code that the operating system returns after each file access is named FILE-STATUS. Both these items are defined later in the DATA DIVISION.

The DATA DIVISION defines all the named data areas to be used in the program.

The FILE SECTION provides an FD (File Description) that specifies 132-character records, and describes the record format in detail. Each record is to consist of the five fields:

CUST-KEY	(30 characters)
ADDR-1, -2 and -3	(lines of the address, 30 characters each field)
TELENO	(the telephone number, 12 characters)

The WORKING-STORAGE SECTION defines all the data areas that are to be reserved for use in the program. The first area is that reserved for the data entry screen, included directly by the COPY statement. The code thus copied is shown exactly as it was generated by the FORMS-2 utility. Notice that COPY statements can be included anywhere as a separate line of a LEVEL II COBOL program to copy in COBOL code from other sources.

The FILE-STATUS data-item already mentioned is then defined as two separate items, STATUS-1 and STATUS-2. The COBOL standard defines these two characters; status character 1 can only be numeric, whereas status character 2 is implementor-defined. The locking mechanism returns the character "D" if a record is locked by another user, and we therefore allow for alphanumeric. The data-item LOCKED is then defined to contain the value "D" for comparison in the locking test routine.

The CURSOR-POSITION item, already mentioned, is then defined as two two-digit numeric fields that will contain the cursor position during execution of the program. These fields are RRCC, the row and column numbers of screen position, respectively.

The level 01 item MESSAGES defines all the messages to be displayed that are not defined as part of the screens generated by FORMS-2.

The MESSAGE-POSITION data-item defines the fixed screen positions (in the coordinates format already described) for direct comparison by the cursor-position detecting routine within the program.

The PROCEDURE DIVISION specifies the procedural part of the program and consists of a series of statements that specify program actions. In this program there are fifteen separate paragraphs, each with a self-explanatory label, e.g. START-PROCEDURE, ENTRY, etc.

These are individually described below:

In the START-PROCEDURE paragraph, the fixed text screen LOKDMO-00 is displayed while the data file is opened. Notice that the file is opened for I-O, which means that if the file does not exist, it will be created. The "file" will consist of two external files, customer and customer.idx, because it was specified as INDEXED SEQUENTIAL. The creation of these files is done automatically by the RTS.

The CHECK-STATUS paragraph is performed next in order to determine whether the file has opened correctly.

In the ENTRY paragraph, the first cursor position in the variable data screen is moved into the CURSOR-POSITION item. This is necessary to indicate that the CURSOR-POSITION variable is to be updated automatically. Its initial value was ZERO, indicating that updating was not required.

The data entry screen LOKDMO-01 is then superimposed on the fixed text screen in order to position the cursor at the first press-selection position. The redefinition of the main screen LOKDMO-00 by LOKDMO-01 and LOKDMO-02 updates the data entered, and any error messages are updated or blanked out as appropriate each time the ENTRY routine is returned to.

The data entry screen is then accepted and, depending on cursor position, the appropriate processing paragraph is entered.

In each of the file processing paragraphs NEXT-RECORD, FIND-RECORD, NEW-RECORD, and UPDATE-RECORD, the action is basically the same. The action-selected message is moved to the display field LOKDMO-01-0011, which is then displayed. The file process is executed, allowing appropriate error conditions, i.e. end of file, invalid key entry, locking and error status checking.

Notice that in the case READ NEXT, the end of file action is specified in the associated START statement, and is specified as invalid key. In the other cases the invalid key condition is specified in the I/O statement itself and arises if the CUST-KEY field is blank or contains data that does not match any of the key fields in the file. In the event of these "errors", control is passed to ERROR-RETRY.

Locking is checked for by comparing status byte 2 with the constant value "D" as stored in the data-item LOCKED. If LOCKED has been returned, control is then passed to RECORD-LOCKED. The status bytes are checked after all I-O processes.

The contents of the file area in memory must be updated either before or after I/O, as relevant. This is done in the paragraphs IN- or OUT-TRANSFER. The IN- and OUT-TRANSFER paragraphs are a series of MOVE statements to move data in or out of the screen data area from or to the file data area. Finally, if successful, the COMPLETED message is displayed.

The FILE-END paragraph defines the action taken if a READ NEXT process encounters end of file. The FAILED message is then displayed and an end of file error message (MESSAGE-2) is displayed beneath the data entry screen as part of screen redefinition LOKDMO-02.

The ERROR-RETRY paragraph displays the FAILED message and the "invalid action" message (MESSAGE-1), returning control to ENTRY; the operator can then retry or change the action.

The CHECK-STATUS paragraph displays the FAILED message and the "disk error" message (MESSAGE-3) and terminates the program via the STOP-IT paragraph. This indicates that a fatal error has occurred. The program terminates and control returns to the operating system.

RECORD-LOCKED displays the LOCKED message (MESSAGE-13) before returning to ENTRY. The user can then try the same access again until the record is unlocked by the other program, or choose another action.

NOT-ACTION blanks out the "action taken" message, displays the "position cursor correctly" message (MESSAGE-12) and returns to ENTRY to repeat the input.

END-IT displays the EXIT selection and drops control into STOP-IT. This closes the file, displays the COMPLETED message and then performs a delaying process so that the operator can see the screen before it is blanked, and before control is returned to the operating system. The delay is achieved by specifying the performance of IN- and OUT-TRANSFER one hundred times in order to prevent the screen from "disappearing" before the operator has noticed and read it.

RUN-TIME ERRORS ORIGINATING IN THE ISAM MODULE

Run-time errors are listed in Appendix E. Note that error 68, "record locked", will arise only if programs are not using STATUS as discussed above. Note also that error 188, "filename too big" will arise if indexed filenames are too long; pathnames may be longer, but the actual basename must be 10 characters or less. ISAM file names are limited to 10 characters on UNIX systems to allow for the addition of the ".idx" suffix.

CHAPTER 10

TERMINAL CONFIGURATION CONSIDERATIONS

OVERVIEW

The use of the LEVEL II COBOL extensions to the ACCEPT and DISPLAY verbs requires knowledge of the particular CRT (VDU) being used. The Run-Time System (RTS) must be able to use terminal capabilities such as cursor addressing and highlighting, for which the terminal control strings vary greatly from one terminal type to another. In addition, the terminal's HOME and cursor control keys (arrow keys, or their equivalents) must be recognized so the proper action can be taken (e.g. to move cursor to the start of the next field).

The RTS has been written so that the user may use ACCEPT and DISPLAY statements with a minimum of special action to interface with the specific terminal being used. In UNIX systems, the terminal type is established from the TERM variable in the user's shell environment, which in turn should be established by the shell during the login process (this is generally done by executing a user's ".profile"). This TERM variable gives the user's terminal type in a well defined and specific code. The RTS uses this code to look up terminal capabilities in the termcap file, which is derived from the University of California at Berkeley /etc/termcap file. If the terminal type is in this file, the RTS will retrieve the required sequences. If the terminal type is not in this file, RTS error 191 will occur.

The user may add terminal types to this file by using the information in the remainder of this chapter and Appendices H and I.

There may be some cases when you wish to use the terminal capabilities set in the local environment (TERMCAP on V7 UNIX systems) as opposed to those in the issued termcap. When this is the case, it is necessary to modify cbrun.c, the C program that initiates program execution. You must modify the portion of the code that nullifies the TERMCAP variable. After the modification is complete, you must "make" and "install" the new version.

TERMCAP FILE

A list of the terminal capability descriptions required by the LEVEL II COBOL RTS is included in Appendix H, and a copy of the UC Berkeley termcap documentation is included in Appendix I. Some systems may already have a /etc/termcap, but this is not always useable, as many of the capabilities (e.g. input sequences for the arrow keys) may not be correctly defined. For this reason, the RTS uses the issued termcap file where the entries are known correct and complete for use with LEVEL II COBOL. All of the terminal capability descriptions used in this file are explained in the Berkeley documentation. If the required capabilities are present in /etc/termcap, the two termcap files may be linked, or the TERMCAP variable may specify an alternate file.

CHAPTER 11

INCORPORATING FORMS-2 UTILITY PROGRAM OUTPUT

INTRODUCTION

The FORMS-2 utility program offers two major facilities to LEVEL II COBOL users:

1. You can define screen layouts to be used in a LEVEL II COBOL application by simply keying the text at the keyboard, and so producing a model form on the screen.

2. You can automatically generate programs to manipulate data input using the created form. In particular, indexed sequential files can be generated and maintained automatically, and these files can be used with LEVEL II COBOL programs.

The FORMS-2 utility is available as a separate software package, and is documented in the FORMS-2 Utility Manual.

SCREEN LAYOUT FACILITIES

The FORMS-2 screen layout facility generates source COBOL record descriptions for screen layouts.

You have three major facilities available:

1. You may store an image copy on disk of the form you have just defined for subsequent use in this or another FORMS-2 run. The image can be printed to obtain a hard copy, using the O/S standard file print utility program.

2. You may generate LEVEL II COBOL source code for the data descriptions required to define the form just created. This may then be included into a LEVEL II COBOL program by use of the COPY verb.

3. You may choose to generate a Check Out program, which allows duplication of many machine conversations that would take place during a run of the application being designed.

All that you have to do to incorporate FORMS-2 screen layout output in a program is to specify the FORMS-2 output file name (filename.DDS) in a COBOL "COPY" statement. Obviously, data item names in the user program must be specified to correspond with those generated from a user-specified base name by FORMS-2. Details of FORMS-2 name generation are given in the FORMS-2 Utility Program User's Guide.

EXAMPLE:

```
000000      COPY     "demo.DDS"
```

GENERATED PROGRAMS

The FORMS-2 utility is capable of generating a COBOL program (the .GEN file option), which maintains data entered in the created forms in an INDEXED SEQUENTIAL file, with automatic generation of file names from a user-supplied base name. These files comply with the standards in use by the operating system under which LEVEL II COBOL is being used.

No special programming is required to use the FORMS-2 generated database. These files can be full maintained interactively using the program generated with the FORMS-2 utility. In addition to creating the ISAM files, this program includes the following facilities:

1. Insertion of new records
2. Insertion of the same data in records with different keys
3. Display of any selected records (full inquiry facility)
4. Insertion or amendment of records dependent on their key
5. Deletion of records
6. Read and display next record or a message if end of file detected
7. Terminate run

Details of the FORMS-2 indexed sequential file handling facilities are given in the FORMS-2 Utility Manual.

APPENDIX A

SUMMARY OF COMPILER DIRECTIVES

Note that where parentheses are specified for use with a directive, they must either be surrounded by quote characters, e.g. "(string)", or escaped, e.g. \(string\), to prevent their interpretation by the shell command line processor. For a detailed explanation of the acceptable formats for these options see Chapter 2.

REFERENCE TABLE OF DIRECTIVES

Table A-1. Reference Table of Directives.

Directive	Use	Default
[NO] ALTER	Allow ALTER statements	ON
[NO] ANIM	Compile suitable for later use with ANIMATOR	ON if ANIMATOR available
[NO] BRIEF	Suppress error messages	OFF
[NO] COMP	Use computational sub-set	OFF
[NO] COPYLIST ["n"]	List COPY files [for segment "n"]	OFF
-c	synonym for COPYLIST	
[NO CRTWIDTH CRTWIDTH "n"]	Set width of CRT to "n"	ON n = 128
DATE "string"	Use "string" for comment-entry in DATE-COMPILED paragraph	ON
[NO] ECHO	Echo errors to console	ON
-k	Synonym for NOECHO	
[NO] ERRLIST	List only errors and flags	OFF
-e	Synonym for ERRLIST	
FILESHARE	Enables file + record locking syntax also enables directives COMMIT(INDEXED) and RESTRICT(INDEXED) to change the default file-type for indexed files.	OFF
[NO FLAG " " FLAG (LOW | L-I | H-I | HIGH | L/II | IBM)]	Flag code higher than level indicated	OFF
[NO FORM FORM "n"]	Suppress headers and form-feeds Set length of page = "n" lines	ON n=60
-f	Synonym for NOFORM	

Directive	Use	Default
[NO INT] [INT "filename"] -i	Specify intermediate code filename Synonym for NOINT	ON = source filename
[NO {LIST} {PRINT} {LIST}["filename"] {PRINT}]	Specify listing requirements	If no directive: ON, i.e. filename = source filename If directive but no filename: filename = standard output
-l	Synonym for LIST":CO:"	
-n	Synonym for NOLIST	
[NO] QUAL	Allows qualified data-names and procedure-names	ON
[NO] REF	Insert addresses on listing	OFF
[NO] RESEQ	Resequence source file	OFF

APPENDIX B

COMPILE-TIME ERRORS

The error numbers and messages as printed by the LEVEL II COBOL compiler are listed below.

ERROR	DESCRIPTION
01	Compiler error; consult Technical Support
02	Illegal format : Data-name
03	Illegal format : Literal, or invalid use of ALL
04	Illegal format : Character
05	Data-name not unique
06	Too many data or procedure names declared
07	Illegal character in column 7 or continuation error
08	Nested COPY statement or unknown COPY file specified
09	'.' missing
10	Statement starts in wrong area of source line
22	DIVISION missing
23	SECTION missing
24	IDENTIFICATION missing
25	PROGRAM-ID missing
26	AUTHOR missing
27	INSTALLATION missing
28	DATE-WRITTEN missing
29	SECURITY missing
30	ENVIRONMENT missing
31	CONFIGURATION missing
32	SOURCE-COMPUTER missing
33	OBJECT-COMPUTER/SPECIAL-NAMES clause error
34	OBJECT-COMPUTER missing
36	SPECIAL-NAMES missing
37	SWITCH clause error or system name/mnemonic name error
38	DECIMAL-POINT clause error
39	CONSOLE clause error
40	Illegal currency symbol
42	DIVISION missing
43	SECTION missing
44	INPUT-OUTPUT missing
45	FILE-CONTROL missing
46	ASSIGN missing
47	SEQUENTIAL or RELATIVE or INDEXED missing
48	ACCESS missing on indexed/relative file
49	SEQUENTIAL or DYNAMIC missing or > 64 alternate keys

50 Illegal ORGANIZATION/ACCESS/KEY combination
51 Unrecognized phrase in SELECT clause
52 RERUN clause syntax error
53 SAME AREA clause syntax error
54 Missing or illegal file-name
55 DATA DIVISION missing
56 PROCEDURE DIVISION missing or unknown statement
57 Program collating sequence not defined
58 EXCLUSIVE, AUTOMATIC or MANUAL missing
59 Non-exclusive lock mode specified for restricted file

62 DIVISION missing
63 SECTION missing
64 File-name not specified in SELECT stmt or invalid CD name
65 RECORD SIZE integer missing or line sequential rec > 1024 bytes
66 Illegal level no (01-49),01 level reqd,or level hierarachy wrong
67 FD, CD or SD qualification syntax error
68 WORKING-STORAGE missing
69 PROCEDURE DIVISION missing or unknown statement
70 Data description qualifier or '.' missing
71 Incompatible PICTURE clause and qualifiers
72 BLANK illegal with non-numeric data-item
73 PICTURE clause too long
74 VALUE with non-elementary item,wrong data-type or value truncated
75 VALUE in error or illegal for PICTURE type
76 Non-elementary item has FILLER/SYNC/JUST/BLANK clause
77 Preceding item at this level has > 8192 bytes or 0 bytes
78 REDEFINES of unequal fields or different levels
79 Data storage exceeds 64K bytes

81 Data description qualifier inappropriate or repeated
82 REDEFINES data-name not declared
83 USAGE must be COMP,DISPLAY or INDEX
84 SIGN must be LEADING or TRAILING
85 SYNCHRONIZED must be LEFT or RIGHT
86 JUSTIFIED must be RIGHT
87 BLANK must be ZERO
88 OCCURS must be numeric, non-zero, unsigned or DEPENDING
89 VALUE must be literal, numeric literal or figurative constant
90 PICTURE string has illegal precedence or illegal char
91 INDFXED data-name missing or already declared
92 Numeric-edited PICTURE string is too large

101 Unrecognized verb
102 IF....ELSE mismatch
103 Operand has wrong data-type or is not declared
104 Procedure name not unique
105 Procedure name same as data-name

106 Name required
107 Wrong combination of data-types
108 Conditional statement not allowed in this context
109 Malformed subscript
110 ACCEPT/DISPLAY wrong or Communications syntax incorrect
111 Illegal syntax used with I-O verb
112 Invalid arithmetic statement
113 Invalid arithmetic expression
114 Compiler error; consult Technical Support
115 Invalid conditional expression
116 IF stmts nested too deep, or too many AFTERs in PERFORM stmt
117 Incorrect structure of PROCEDURE DIVISION
118 Reserved word missing or incorrectly used
119 Too many subscripts in one statement
120 Too many operands in one statement
121 LOCK clause specified for file with lock mode EXCLUSIVE
122 KEPT specified for uncommittable file
123 KEPT omitted for committable file

141 Inter-segment procedure name duplication
142 IF....ELSE mismatch at end of source input
143 Operand has wrong data-type or not declared
144 Procedure name undeclared
145 INDEX data-name declared twice
146 Bad cursor control : illegal AT clause
147 KEY declaration missing or illegal
148 STATUS declaration missing
149 Bad STATUS record
150 Undefined inter-segment reference or error in ALTERed para
151 PROCEDURE DIVISION in error
152 USING parameter not declared in LINKAGE SECTION
153 USING parameter not level 01 or 77
154 USING parameter used twice in parameter list
155 FD missing

157 Incorrect structure of PROCEDURE DIVISION

160 Too many operands in one statement

APPENDIX C

SUMMARY OF NATIVE CODE GENERATOR DIRECTIVES

Note that where parentheses are used with a directive, they must either be surrounded by quotes (e.g. "(string)") or escaped (e.g. (string), to prevent their interpretation by the UNIX shell. For a detailed description of the acceptable formats see Chapter 3.

REFERENCE TABLE OF DIRECTIVES

Directive	Use	Default
[NO] ASM	Specify assembly listing requirements	OFF
-a	Synonym for ASM	
[NO] BELL	Suppress or request finish bleep	ON
BELL "n"	Define n = Bell character	7
[NO] CHECK	Specify checking of run-time limit violations	ON
-c	Synonym for NOCHECK	
[NO] FORM	Suppress listing headers and form feeds	ON n=60
FORM "n"	Let length of page = "n" lines	
-f	Synonym for NOFORM	
[NO] GNT	Suppress code generation output file	ON
GNT "filename"	Specify file for generated code	
-g	Synonym for NOGNT	
[NO] LIST	Specify listing requirements	ON
LIST [(destination) "destination"]		
-l	Synonym for LIST":CO:"	
-n	Synonym for NOLIST	
PAGETHROW "n"	Specify ASCII character code for physical printer page throw	12

APPENDIX D

CODE GENERATION ERRORS

Error Number	Message	Notes
0	Illegal error number	Call to error procedure with unknown error number
1	Illegal intermediate code	Intermediate code operation invalid - input file may corrupt
2	Site setup bad index	Initialisation of optimisation tables. Same leading code occurs twice.
4	Site setup bad code	Initialisation of optimisation tables. Code value out of range.
5	Invalid NK descriptor	Numeric operation with invalid constant descriptor.
6	Dynamic storage overflow	May be caused by attempt to generate a program that is too large or complex, or by limitations of memory.
7	ISR with invalid segment	Inter-segment reference file has segment number that is not zero or 50 to 99. ISR file may be corrupt.
8	Unprocessed transient code	A transient code has been left unprocessed. Usually caused by errors or missing rules in optimisation tables.
9	Expected symbol not found	Dictionary (dynamic memory) search fails to find item that should be present.
10	Input file is generated Code	Code generator output used as input.
11	Generated code exceeds 64K bytes	Attempt to generate too large output program without artificial segmentation feature available.

12	Illegal action number	Action routine specified in tables but not provided in action routine procedure.
13	Condition stack overflow	IF statements nested too deeply in COBOL program. Limit is 64 nested conditions.
14	Input file not intermediate code or wrong version number	The input file specified does not contain intermediate code, or the code generator version differs from that of compiler used to produce the intermediate code. May be caused by the incorrect use of the VERSION command.
15	Too many segment breaks	Automatic segmentation allows 10 output segments per input segment. Error 15 means this limit has been exceeded. A possible solution is to divide the input into more and smaller segments using COBOL segmentation.
16	Too many output segments	Automatic segmentation allows up to 200 generated output segment files. Error 16 means that this limit has been exceeded.

APPENDIX E

RUN-TIME ERRORS

RUN TIME ERRORS

Error	Description
001	Not owner of file
002	No such file
003	Too many ISAM files, or no such process
005	Physical I/O error
009	Incorrect mode or file descriptor
013	Attempt to access a file with incorrect permission
017	File already exists
021	File is a directory
024	Too many open files
028	No space on device
030	File system is read only
065	ISAM - File locked
066	ISAM - Attempted to add duplicate record key
067	ISAM - File not open
068	ISAM - Record locked
069	ISAM - Illegal argument to ISAM module
070	ISAM - Too many files open
071	ISAM - Bad ISAM file format
072	ISAM - End of file
073	ISAM - No record found
074	ISAM - No current record
075	ISAM - Data file name too long
076	ISAM - Can't create lock file in /isam
077	ISAM - Internal ISAM error
078	ISAM - Illegal key descriptor
079	ISAM - Is primary key
080	ISAM - Non-exclusive access
081	ISAM - Key already exists
150	ISAM - ANIMATOR load error
151	Random read on sequential file
152	Rewrite on file not opened I/O
153	Subscript out of range
154	Perform nesting exceeds 22 levels
155	Illegal command line
156	Invalid file operation
157	Object file too large
158	Rewrite on line sequential file
159	Malformed line sequential file
160	Overlay loading error
161	Illegal intermediate code
162	Arithmetic overflow or underflow
164	Specified call code not supplied

165 Incompatible releases of compiler and RTS
166 Attempt to open file which is already open
170 Illegal operation in indexed sequential
172 Attempt to call active program
173 Intermediate code file not found
174 Intersegment reference table not found
180 COBOL file malformed
181 Fatal file malformation
182 Open CI or CO in wrong direction
183 Attempt to open LS file for I-O
184 Accept/Display I-O error
185 Can't load COBOL RTS module
186 Internal RTS error - should not occur!
187 Version 3 subset error
188 Filename too big
189 Intermediate code load error
190 Too many arguments to CALL
191 Terminal type not defined, or not in termcap file
192 Required terminal capability description missing
193 Null file name used in a file operation
194 Memory allocation error
195 Dictionary error
196 Not implemented in this RTS release

ERROR REPORTING

When the RTS detects a fatal error, it prints out the general category of error along with an associated filename; e.g. "Load error on file pi.INT". It then prints out the error code and the current pc. Finally, it looks up the error code in a system error file (/usr/lib/cobol/cblerrs) and prints out the associated message; these are identical to those in the above table. There are two types of run-time errors: recoverable and fatal.

* Fatal errors

  Fatal errors may be signalled from the operating system or from the RTS. Fatal errors cause a message to be output to the console which includes a three-digit error code and reference to the COBOL statement in which it occurred. These fall into two classes:

  (i) Exceptions: these cover arithmetic overflow, subscript out of range, too many levels of perform nesting.

  (ii) I-O errors: input-output errors for which STATUS has not been selected.

* Recoverable errors

  If the COBOL programmer has selected STATUS for a file then it is assumed that any non-fatal errors will be handled by the application program, and no action is taken. If the programmer has not selected STATUS, the program will exit. The program overleaf indicates a method by which STATUS can be tested.

APPENDIX F

OPERATING SYSTEM ERRORS

These errors appear in the same format as the LEVEL II COBOL errors. Conventionally, error numbers 1-64 are reserved for the operating system. If an error appears that is not in the following list, it may be looked up in Volume 1, section II of the UNIX Programmer's Manual.

Users of other LEVEL II COBOL systems should note that the UNIX error return for a non-existent file is 2, and that programs that check for a specific error return from another operating system may not work correctly.

Error	Description
1	Not owner of file
2	No such file
3	Too many ISAM files, or no such process
5	Physical I/O error
9	Incorrect mode or file descriptor
13	Attempt to access a file with incorrect permission
17	File already exists
21	File is a directory
24	Too many open files
28	No space on device
30	File system is read-only

APPENDIX G

INTERACTIVE DEBUG COMMANDS

COMMAND	EFFECT
A data-ref val	Change value at address given to val (data division)
B data-ref	Execute until data-ref changes.
C val	Display ASCII character corresponding to val
D data-ref	Display 16 bytes from address given
D,	Display next 16 bytes
E data-ref val	Execute until data-ref equals val. (data division)
G proc-ref	Execute from current position until address reached
L	Output carriage return/line feed to console
M name	Start definition of macro
P	Display current program counter
S data-ref	Set work register to address given
T proc-ref	Trace all paragraphs executed up to address
X	Execute one instruction
$	End macro definition
/	Display byte at address in work register
. val	Change byte at address in work register to val and increment register
,	Increment work register
;	Start comment - line up to carriage return

where:

data-ref	-	16 bit hex value (4 digits) in data area
proc-ref	-	16 bit hex value (4 digits) in code area
val	-	8 bit value (2 hex digits) or double quotes and an ASCII char (e.g. "A)
name	-	single ASCII character in code area

APPENDIX H

TERMINAL CONFIGURATION ISSUES

The Run-Time System (RTS) will configure itself at run time for your terminal. The UC Berkeley termcap format and termlib accessing routines are used to accomplish this (See Appendix I). The terminal capability data that are required by the RTS reside in file /usr/lib/cobol/termcap.

Here we list the capability entries used and/or required by the RTS, along with any assumptions made about terminal characteristics.

Where terminal control characteristics are variable, either by switch settings or by the input of certain control sequences, care must be taken to ensure the correspondence between these settings and the termcap file entries.

REQUIRED ENTRIES

* li - the number of lines on the terminal.
* co - the number of columns on the terminal.
* bs or bc - bs if terminal backspaces with CTRL-H, bc if a different sequence is required.
* up - sequence to move cursor up one line, same column.
* cl - sequence to clear screen, cursor left at home.
* cm - sequence to move cursor to an addressed screen location.
* kl - sequence sent by left-arrow key.
* kr - sequence sent by right-arrow key.
* ku - sequence sent by up-arrow key.
* kd - sequence sent by down-arrow key.
* kh - sequence sent by home key.

OPTIONAL ENTRIES

* so - sequence to turn on "hiliting", (e.g. reverse video).

* se - sequence to turn off "hiliting".

* sg - number of screen positions taken by the "hiliting" sequences.

ASSUMPTIONS

* the sequence sent by the return key is CTRL-M.

* the sequence sent by the tab key is CTRL-I.

* the sequence to ring the bell or beeper is CTRL-G.

APPENDIX I

UC BERKELEY TERMCAP

NAME

termcap - terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals. Terminals are described in termcap by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in termcap.

Entries in termcap consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same

cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give "ei=:" if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give "im=:" if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)

nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use cm
ti	str		String to begin programs that use cm
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overstrike
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like ce \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

The following entry, which describes the Concept-100, is among the more complex entries in the termcap file as of this writing. (Note that this particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200:\
    :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
    :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=:\
    :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Note that entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in termcap are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability am. Hence the description of the Concept includes am. Numeric capabilities are followed by the character '#' and then the value. Thus co which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as ce (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'.

A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A \E maps to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences \n \r \t \b \f give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \ , and the characters ^ and \ may be given as \^ and \\ . If it is necessary to place a : in a capability it must be escaped in octal as \072. If it is necessary to place a null character in a string capability it must be encoded as \200. The routines which deal with termcap use C strings, and strip the high bits of the output very late so that a \200 comes out as a \000 would.

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in termcap and to build up a description gradually, using partial descriptions with ex to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the termcap file to describe it or bugs in ex. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in termcap. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor. (This only works on version 7 systems.)

BASIC CAPABILITIES

The number of columns on each line for the terminal is given by the co numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the li capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the am capability. If the terminal can clear its screen, then this is given by the cl string capability. If the terminal can backspace, then it should have the bs capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the bc string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the os capability.

A very important point here is that the local cursor motions encoded in termcap are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the am capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the termcap file usually assumes that this is on, i.e. am.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

CURSOR ADDRESSING

Cursor addressing in the terminal is described by a cm string capability, with printf(3s) like escapes %x in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the cm string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the % encodings have the following meanings:

```
%d   as in printf, 0 origin
%2   like %2d
%3   like %3d
%.   like %c
%+x  adds x to value, then %.
%>xy if value > x adds y, no output.
%r   reverses order of line and column, no output
%i   increments line/column (for 1 origin)
%%   gives a single %
%n   exclusive or row and column with 0140 (DM2500)
%B   BCD (16*(x/10)) + (x%10), no output.
%D   Reverse coding (x-2*(x%16)), no output. (Delta Data).
```

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its cm capability is "cm=6 \E&%r%2c%2Y" The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "cm=^T%.%." Terminals which use " need to be able to backspace the cursor (bs or bc), and to move the cursor up one line on the screen (up introduced below). This is necessary because it is not always safe to transmit \t, \n^D and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "cm=\E=%+ %+ "

CURSOR MOTIONS

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as nd (non- destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as up. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as ho; similarly a fast way of getting to the lower left hand corner can be given as ll; this may involve going up with up from the home position, but the editor will never do this itself (unless ll does) because it makes no assumption about the effect of moving up from the home position.

AREA CLEARS

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as ce. If the terminal can clear from the current position to the end of the display, then this should be given as cd. The editor only uses cd from the first column of a line.

INSERT/DELETE LINE

If the terminal can open a new blank line before the line where the cursor is, this should be given as al; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as dl; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as sb, but just al suffices. If the terminal can retain display memory above then the da capability should be given; if display memory can be retained below then db should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with sb may bring down non-blank lines.

INSERT/DELETE CHARACTER

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using termcap. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def" Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability in, which stands for "insert null" If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as im the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as ei the sequence to leave insert mode (give this, with an empty value also if you gave im so). Now give as ic any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give ic, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.)
If post insert padding is needed, give this as a number of milliseconds in ip (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in ip. It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability mi to speed up inserting in this case. Omitting mi will affect only speed. Some terminals (notably Datamedia's) must not have mi because of the way their insert mode works.

Finally, you can specify delete mode by giving dm and ed to enter and exit delete mode, and dc to delete a single character while in delete mode.

HIGHLIGHTING, UNDERLINING, AND BELLS

If your terminal has sequences to enter and exit standout mode these can be given as so and se respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining - half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the prefered mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

Codes to begin underlining and end underlining can be given as us and ue respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as uc. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as vb; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of ex, this can be given as vs and ve, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as ti and te. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability ul. If overstrikes are erasable with a blank, then this should be indicated by giving eo.

KEYPAD

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as ks and ke. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as kl, kr, ku, kd, and kh respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as k0, k1, ..., k9. If these keys have labels other than the default f0 through f9, the labels can be given as 10, 11, ..., 19. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the termcap 2 letter codes can be given in the ko capability, for example, "ko=cl,ll,sf,sb:" which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The ma entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with kl, kr, ku, kd, and kh. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are h for kl, j for kd, k for ku, l for kr, and H for kh. For example, the mime would be :ma=^Kj^Zk^Xl: indicating arrow keys left (^H), down (^K), up (^Z), and right (^X).

MISCELLANEOUS CONSIDERATIONS

If the terminal requires other than a null (zero) character as a pad, then this can be given as pc.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as ta.

Hazeltine terminals, which don't allow tilde characters to be printed should indicate hz. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate nc. Early Concept terminals, which ignore a linefeed immediately after an am wrap, should indicate xn. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), xs should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate xt. Other specific terminal problems may be corrected by adding more capabilities of the form x.

Other capabilities include is, an initialization string for the terminal, and if, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, is will be printed before if. This is useful where if is /usr/lib/tabset/std but is clears the tabs first.

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability tc can be given with the name of the similar terminal. This capability must be last and the combined length of the two entries must not exceed 1024. Since termlib routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with xx@ where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the ks or ke capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES /etc/termcap file containing terminal descriptions

BUGS

Ex allows only 256 characters for string capabilities, and the routines in termcap do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024. The ma, vs, and ve entries are specific to the vi program. Not all programs support all entries. There are entries that are not supported by any program.

APPENDIX J

LEVEL II COBOL IN THE UNIX ENVIRONMENT

THE CONSOLE

In LEVEL II COBOL, and throughout this manual, the term "console" refers to the user's terminal, not to the system console as might be expected by those more familiar with UNIX usage.

DEVICES

Built into the compiler and Run-Time System (RTS) is the concept of certain devices. You may write a program that will read from the Console In device (":CI:"), or write to the Console Out or Error devices (":CO:" and ":CE:"). A COBOL program running on UNIX that uses these devices as file names will access the standard input and output, and error output, respectively, since these are connected to your terminal, unless redirected. In addition, the file name ":LP:" is recognized as a printer-type file; if this is used as the name of a sequential file all writes will cause printer-type carriage control to be applied to the records. As discussed in Chapter 4, if the file name ":CI:" is declared ORGANIZATION LINE SEQUENTIAL, then the first :CI: read in the program will access the command line arguments. Note that ACCEPT FROM CONSOLE will do the same, but ACCEPT FROM CRT will not.

DATA FILES

ANSI COBOL defines the concepts of sequential, relative, and indexed sequential files, all containing fixed length records. To this, LEVEL II COBOL adds a line sequential (LS) file type, which is variable length records in standard UNIX text file format. LS files (such as LEVEL II COBOL source and listing files) consist of lines of text each terminated with a newline character.

Sequential files consist of fixed length records with no terminators; the lack of newline characters (unless they have been explicitly placed there by the COBOL program) means that UNIX utilities such as grep and sort may not work on sequential type files. Relative files consist of fixed length records with a 1-byte terminator; if a particular numbered record is present in the file, this terminator will be a newline character. If a record is logically not present in the file, the record and terminator will be all nulls (ASCII 0). COBOL relative files must not be changed by any utility that removes nulls, such as the editor.

COMMAND LINES AND SPECIAL CHARACTERS

As already discussed in Chapter 2, certain characters may be important to the shell or command line handler on your system. Consequently, these must be masked when used in command lines. See the shell or command line handler documentation for your system for further details.

CHANGE OF TERMINAL MODE FOR LEVEL II ACCEPT/DISPLAY

Normal terminal interfacing is performed by the UNIX operating system in a mode referred to as "cooked". It is in this mode that the operating system interprets such terminal functions as Interrupt, Quit, CTRL-D (EOF), CTRL-S (XOFF), CTRL-Q (XON), Tab, etc.

When a COBOL application program calls for ACCEPT or DISPLAY UPON CRT, the COBOL run-time system takes more control over tty handling. The RTS sets the terminal from "cooked" mode to a special mode. Look in the LEVEL II COBOL Language Reference Manual for more information on ACCEPT/DISPLAY UPON CRT statements. Some "cooked" terminal functions such as those mentioned above may no longer operate, or may only operate in certain cases. For example, a CTRL-D (normally interpreted by the UNIX operating system as an end of file) is not defined in the Language Reference Manual, and is therefore not interpreted by the RTS.

On V7 systems the following terminal characters are processed:

Pause (XOFF)	CTRL-S
Continue (XON)	CTRL-Q
Interrupt	Rubout (or Delete)
Quit	CTRL-\ or CTRL-SHIFT-L

When a COBOL application program calls for ACCEPT or DISPLAY UPON CONSOLE (after there has already been a call to ACCEPT or DISPLAY UPON CRT) RTS routines attempt to simulate UNIX "cooked" processing. Thus, terminal functions work as they do with the UNIX operating system. This RTS simulation, however, has two major differences:

1. Tabs are expanded to one space instead of to one tab stop as by the UNIX operating system.

2. No lower-to-upper case mapping, as occurs for "stty lcase".

TYPE-AHEAD

Type-ahead refers to characters entered on the terminal that have not yet been read by a program. Note that while a COBOL program is running, any type-ahead will be flushed upon the first execution of an ACCEPT or DISPLAY UPON CRT statement, so in general type ahead should be avoided before the first ACCEPT FROM CRT.

OVERFLOW OF MEMORY

Several error messages (for example, the failure to load a CALL'ed module) may be the result of an overflow of the allocated memory area, although the message may not state this explicitly. You should suspect this problem whenever a large but otherwise correct intermediate code file fails to load.

APPENDIX K

LOCKDEMO SOURCE CODE

The source code for the COBOL program "lockdemo" is reproduced below; this includes lockdemo.DDS, which was generated by the FORMS-2 utility program.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.   LOCKING-TEST.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL-NAMES.
         CONSOLE IS CRT
         CURSOR IS CURSOR-POSITION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
         SELECT CUSTOMER-FILE ASSIGN "customer"
                  ORGANIZATION INDEXED
                  ACCESS DYNAMIC
                  RECORD KEY CUST-KEY
                  STATUS FILE-STATUS.

       DATA DIVISION.
       FILE SECTION.
       FD  CUSTOMER-FILE;   RECORD 132.
       01  CUSTOMER-RECORD.
           03   CUST-KEY   PIC X(0030).
           03   CUST-DAT.
              05   ADDR-1  PIC X(0030).
              05   ADDR-2  PIC X(0030).
              05   ADDR-3  PIC X(0030).
              05   TELENO  PIC 9(0012).
       WORKING-STORAGE SECTION.
       COPY "lockdemo.DDS".
       01  FILE-STATUS.
           02   STATUS-1   PIC 9.
           02   STATUS-2   PIC X.
       01  LOCKED          PIC X      VALUE "D".
       01  CURSOR-POSITION PIC 9(4)   VALUE 0.
       01  MESSAGES.
           03   MESSAGE-1  PIC X(51)  VALUE "INVALID ACTION: UPDATE NEW
      -    "OR ENTER EXISTING RECORD".
           03   MESSAGE-2  PIC X(41)  VALUE "END OF FILE - FIND, ENTER O
      -    "R EXIT OPTIONS".
           03   MESSAGE-3  PIC X(39)  VALUE "DISK ERROR - EXITING".
           03   MESSAGE-4       PIC X(6)   VALUE " NEXT ".
           03   MESSAGE-5       PIC X(6)   VALUE " FIND ".
           03   MESSAGE-6       PIC X(6)   VALUE " NEW  ".
```

```
     03   MESSAGE-7        PIC X(6)   VALUE "UPDATE".
     03   MESSAGE-8        PIC X(6)   VALUE " EXIT ".
     03   MESSAGE-9        PIC X(9)   VALUE "COMPLETED".
     03   MESSAGE-10       PIC X(9)   VALUE " LOCKED  ".
     03   MESSAGE-11       PIC X(9)   VALUE " FAILED  ".
     03   MESSAGE-12       PIC X(53)  VALUE "POSITION CURSOR AT ACT
-    "ION POSITION AND TRY AGAIN".
     03   MESSAGE-13       PIC X(43)  VALUE "RECORD LOCKED - FIND,
-    "ENTER OR EXIT OPTIONS".
 01  MESSAGE-POSITION.
     03   ACTION-POS-1     PIC 9(4)   VALUE 0801.
     03   ACTION-POS-2     PIC 9(4)   VALUE 0817.
     03   ACTION-POS-3     PIC 9(4)   VALUE 0832.
     03   ACTION-POS-4     PIC 9(4)   VALUE 0848.
     03   ACTION-POS-5     PIC 9(4)   VALUE 0865.
 PROCEDURE DIVISION.
 START-PROCEDURE.
     DISPLAY SPACE.
     DISPLAY LOKDMO-00.
     OPEN I-O CUSTOMER-FILE.
     PERFORM CHECK-STATUS.
 ENTRY.
     MOVE ACTION-POS-1 TO CURSOR-POSITION.
     DISPLAY LOKDMO-01.
     ACCEPT LOKDMO-01.
     MOVE SPACES TO LOKDMO-02-0001, LOKDMO-02-0002.
     DISPLAY LOKDMO-02.
     IF CURSOR-POSITION EQUAL ACTION-POS-2 GO TO FIND-RECORD.
     IF CURSOR-POSITION EQUAL ACTION-POS-3 GO TO NEW-RECORD.
     IF CURSOR-POSITION EQUAL ACTION-POS-4 GO TO UPDATE-RECORD.
     IF CURSOR-POSITION EQUAL ACTION-POS-5 GO TO END-IT.
     IF CURSOR-POSITION GREATER ACTION-POS-5 GO TO NOT-ACTION.
 NEXT-RECORD.
     MOVE MESSAGE-4 TO LOKDMO-01-0011.
     DISPLAY LOKDMO-01.
     START CUSTOMER-FILE KEY GREATER THAN CUST-KEY
        INVALID GO TO FILE-END.
     READ CUSTOMER-FILE NEXT.
     IF STATUS-2 EQUAL LOCKED GO TO RECORD-LOCKED.
     PERFORM CHECK-STATUS.
     PERFORM IN-TRANSFER.
     DISPLAY LOKDMO-01.
     MOVE MESSAGE-9 TO LOKDMO-02-0001.
     DISPLAY LOKDMO-02.
     GO TO ENTRY.
 FIND-RECORD.
     PERFORM CLEAR-FIELDS.
     MOVE MESSAGE-5 TO LOKDMO-01-0011.
     DISPLAY LOKDMO-01.
     MOVE LOKDMO-01-0006 TO CUST-KEY.
     READ CUSTOMER-FILE INVALID GO TO ERROR-RETRY.
```

```
    IF STATUS-2 EQUAL LOCKED GO TO RECORD-LOCKED.
    PERFORM CHECK-STATUS.
    PERFORM IN-TRANSFER.
    DISPLAY LOKDMO-01.
    MOVE MESSAGE-9 TO LOKDMO-02-0001.
    DISPLAY LOKDMO-02.
    GO TO ENTRY.
NEW-RECORD.
    MOVE MESSAGE-6 TO LOKDMO-01-0011.
    DISPLAY LOKDMO-01.
    PERFORM OUT-TRANSFER.
    WRITE CUSTOMER-RECORD INVALID GO TO ERROR-RETRY.
    IF STATUS-2 EQUAL LOCKED GO TO RECORD-LOCKED.
    PERFORM CHECK-STATUS.
    MOVE MESSAGE-9 TO LOKDMO-02-0001.
    PERFORM CLEAR-NAME THROUGH CLEAR-FIELDS.
    DISPLAY LOKDMO-02.
    GO TO ENTRY.
UPDATE-RECORD.
    MOVE MESSAGE-7 TO LOKDMO-01-0011.
    DISPLAY LOKDMO-01.
    PERFORM OUT-TRANSFER.
    REWRITE CUSTOMER-RECORD INVALID GO TO ERROR-RETRY.
    IF STATUS-2 EQUAL LOCKED GO TO RECORD-LOCKED.
    PERFORM CHECK-STATUS.
    MOVE MESSAGE-9 TO LOKDMO-02-0001.
    DISPLAY LOKDMO-02.
    PERFORM CLEAR-NAME THROUGH CLEAR-FIELDS.
    GO TO ENTRY.
IN-TRANSFER.
    MOVE CUST-KEY TO LOKDMO-01-0006.
    MOVE ADDR-1 TO LOKDMO-01-0007.
    MOVE ADDR-2 TO LOKDMO-01-0008.
    MOVE ADDR-3 TO LOKDMO-01-0009.
    MOVE TELENO TO LOKDMO-01-0010.
OUT-TRANSFER.
    MOVE LOKDMO-01-0006 TO CUST-KEY.
    MOVE LOKDMO-01-0007 TO ADDR-1.
    MOVE LOKDMO-01-0008 TO ADDR-2.
    MOVE LOKDMO-01-0009 TO ADDR-3.
    MOVE LOKDMO-01-0010 TO TELENO.
CLEAR-NAME.
    MOVE SPACE TO LOKDMO-01-0006.
CLEAR-FIELDS.
    MOVE SPACE TO LOKDMO-01-0007.
    MOVE SPACE TO LOKDMO-01-0008.
    MOVE SPACE TO LOKDMO-01-0009.
    MOVE SPACE TO LOKDMO-01-0010.
FILE-END.
    MOVE MESSAGE-11 TO LOKDMO-02-0001.
    MOVE MESSAGE-2 TO LOKDMO-02-0002.
```

```
           DISPLAY LOKDMO-02.
           PERFORM CLEAR-NAME THROUGH CLEAR-FIELDS.
           MOVE ZERO TO CUST-KEY.
           GO TO ENTRY.
       ERROR-RETRY.
           MOVE MESSAGE-11 TO LOKDMO-02-0001.
           MOVE MESSAGE-1 TO LOKDMO-02-0002.
           DISPLAY LOKDMO-02.
           GO TO ENTRY.
       CHECK-STATUS.
           IF STATUS-1 NOT EQUAL ZERO
              MOVE MESSAGE-11 TO LOKDMO-02-0001
              MOVE MESSAGE-3 TO LOKDMO-02-0002
              DISPLAY LOKDMO-02
              GO TO STOP-IT.
       RECORD-LOCKED.
           MOVE MESSAGE-10 TO LOKDMO-02-0001.
           MOVE MESSAGE-13 TO LOKDMO-02-0002.
           DISPLAY LOKDMO-02.
           PERFORM CLEAR-FIELDS.
           GO TO ENTRY.
       NOT-ACTION.
           MOVE SPACES TO LOKDMO-02-0001.
           MOVE MESSAGE-12 TO LOKDMO-02-0002.
           DISPLAY LOKDMO-02.
           GO TO ENTRY.
       END-IT.
           MOVE MESSAGE-8 TO LOKDMO-01-0011.
           DISPLAY LOKDMO-01.
       STOP-IT.
           CLOSE CUSTOMER-FILE.
           MOVE MESSAGE-9 TO LOKDMO-02-0001.
           DISPLAY LOKDMO-02.
           PERFORM IN-TRANSFER THRU OUT-TRANSFER 100 TIMES.
           DISPLAY SPACE.
           STOP RUN.
```

LOCKDEMO.DDS

```
       01  LOKDMO-00    .
           03 FILLER              PIC X(0184).
           03 LOKDMO-00-0001      PIC X(0031) VALUE "LOCKING DEMONSTRATION
      -    "PROGRAM  ".
           03 FILLER              PIC X(0105).
           03 LOKDMO-00-0002      PIC X(0074) VALUE "This program demonstra
      -    "tes the use of record locking facilities using an   ".
           03 FILLER              PIC X(0006).
           03 LOKDMO-00-0003      PIC X(0030) VALUE "ISAM file and I-O acce
      -    "ss mode.".
```

```
           03 FILLER           PIC X(0002).
           03 LOKDMO-00-0004   PIC X(0035) VALUE "You can specify action
      -    "s as follows:".
           03 FILLER           PIC X(0093).
           03 LOKDMO-00-0005   PIC X(0013) VALUE "* NEXT record".
           03 FILLER           PIC X(0003).
           03 LOKDMO-00-0006   PIC X(0013) VALUE "* FIND record".
           03 FILLER           PIC X(0002).
           03 LOKDMO-00-0007   PIC X(0014) VALUE "* ENTER record".
           03 FILLER           PIC X(0002).
           03 LOKDMO-00-0008   PIC X(0015) VALUE "* UPDATE record".
           03 FILLER           PIC X(0002).
           03 LOKDMO-00-0009   PIC X(0006) VALUE "* EXIT".
           03 FILLER           PIC X(0018).
           03 LOKDMO-00-0010   PIC X(0060) VALUE "(Position cursor over
      -    "appropriate asterisk and press RETURN)".
           03 FILLER           PIC X(0092).
           03 LOKDMO-00-0011   PIC X(0080) VALUE "----------------------
      -    "------------------------------------------------------------".
           03 FILLER           PIC X(0009).
           03 LOKDMO-00-0012   PIC X(0005) VALUE "NAME:".
           03 FILLER           PIC X(0006).
           03 LOKDMO-00-0013   PIC X(0001) VALUE "[".
           03 FILLER           PIC X(0030).
           03 LOKDMO-00-0014   PIC X(0001) VALUE "]".
           03 FILLER           PIC X(0037).
           03 LOKDMO-00-0015   PIC X(0008) VALUE "ADDRESS:".
           03 FILLER           PIC X(0003).
           03 LOKDMO-00-0016   PIC X(0001) VALUE "[".
           03 FILLER           PIC X(0030).
           03 LOKDMO-00-0017   PIC X(0001) VALUE "]".
           03 FILLER           PIC X(0048).
           03 LOKDMO-00-0018   PIC X(0001) VALUE "[".
           03 FILLER           PIC X(0030).
           03 LOKDMO-00-0019   PIC X(0001) VALUE "]".
           03 FILLER           PIC X(0048).
           03 LOKDMO-00-0020   PIC X(0001) VALUE "[".
           03 FILLER           PIC X(0030).
           03 LOKDMO-00-0021   PIC X(0001) VALUE "]".
           03 FILLER           PIC X(0037).
           03 LOKDMO-00-0022   PIC X(0012) VALUE "TELEPHONE: [".
           03 FILLER           PIC X(0012).
           03 LOKDMO-00-0023   PIC X(0001) VALUE "]".
           03 FILLER           PIC X(0046).
           03 LOKDMO-00-0024   PIC X(0080) VALUE "----------------------
      -    "------------------------------------------------------------".
           03 FILLER           PIC X(0089).
           03 LOKDMO-00-0025   PIC X(0016) VALUE "(Last action was".
           03 FILLER           PIC X(0007).
           03 LOKDMO-00-0026   PIC X(0008) VALUE ", and it".
           03 FILLER           PIC X(0010).
```

```
    03 LOKDMO-00-0027   PIC X(0001) VALUE ")".
01  LOKDMO-01    REDEFINES LOKDMO-00    .
    03 FILLER           PIC X(0560).
    03 LOKDMO-01-0001   PIC *.
    03 FILLER           PIC X(0015).
    03 LOKDMO-01-0002   PIC *.
    03 FILLER           PIC X(0014).
    03 LOKDMO-01-0003   PIC *.
    03 FILLER           PIC X(0015).
    03 LOKDMO-01-0004   PIC *.
    03 FILLER           PIC X(0016).
    03 LOKDMO-01-0005   PIC *.
    03 FILLER           PIC X(0276).
    03 LOKDMO-01-0006   PIC X(0030).
    03 FILLER           PIC X(0050).
    03 LOKDMO-01-0007   PIC X(0030).
    03 FILLER           PIC X(0050).
    03 LOKDMO-01-0008   PIC X(0030).
    03 FILLER           PIC X(0050).
    03 LOKDMO-01-0009   PIC X(0030).
    03 FILLER           PIC X(0050).
    03 LOKDMO-01-0010   PIC 9(0012).
    03 FILLER           PIC X(0233).
    03 LOKDMO-01-0011   PIC X(0006).
01  LOKDMO-02    REDEFINES LOKDMO-00    .
    03 FILLER           PIC X(1481).
    03 LOKDMO-02-0001   PIC X(0009).
    03 FILLER           PIC X(0110).
    03 LOKDMO-02-0002   PIC X(0071).
```