HP COBOL II/XL Programmer's Guide

900 Series HP 3000 Computer Systems

HP Part No. 31500-90002
Printed in U.S.A. July 1991

E0791

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Printing History

New editions are complete version of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title pages change only when a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	November 1987	31500A.00.12
Second Edition	October 1988	31500A.01.06
Third Edition	July 1991	31500A.04.03

Preface

This *HP COBOL II/XL Programmer's Guide* for the Hewlett-Packard HP COBOL II/XL programming language is intended for experienced COBOL programmers who are familiar with the MPE XL file system and HP 3000 subsystems. It discusses selected HP COBOL II/XL topics in detail and explains statement interaction where necessary. It does not explain every feature of HP COBOL II/XL, as the *HP COBOL II/XL Reference Manual* does.

The following briefly describes each chapter.

Chapter 1 Describes HP COBOL II/XL in terms of its relationship to COBOL II, ANSI COBOL 1985, ANSI COBOL 1974, and its environment.

Chapter 2 Describes HP COBOL II/XL in terms of its features.

Chapter 3 Explains structured programming. Tells you how to design your program for maximum run-time efficiency and portability.

Chapter 4 Explains how your program can call subprograms and intrinsics.

Chapter 5 Explains how your program can use files.

Chapter 6 Explains how to compile and link your program.

Chapter 7 Explains how to debug your program.

Additional Documentation

Refer to the following manuals for more information about HP COBOL II/XL:

- *HP COBOL II/XL Reference Manual* (31500-90001)
- *HP COBOL II/XL Quick Reference Guide* (31500-90003)

Refer to the following manual for information about migrating HP COBOL II/V programs to HP COBOL II/XL:

- *HP COBOL II/XL Migration Guide* (31500-90004)

This manual references the following manuals:

- *HP FORTRAN 77 Programmer's Guide* (5957-4686)
- *KSAM/3000 Reference Manual* (30000-90079)
- *Using KSAM/XL* (32650-90168)
- *Using Files: A Guide for New Users of HP 3000 Computer Systems* (30000-90102)
- *TurboIMAGE/XL Reference Manual* (30391-90001)
- *HP Pascal/XL Reference Manual* (31502-90001)
- *HP Pascal/XL Programmer's Guide* (31502-90002)
- *HP System Dictionary/XL General Reference Manual* (32256-90004)
- *HP Screen Management Intrinsic Library Reference Manual* (32424-90002)
- *MPE XL Commands Reference Manual* (32650-90003)
- *MPE XL Intrinsics Reference Manual* (32650-90028)
- *MPE XL Error Message Manual, Volume 1* (32650-90066)
- *MPE XL Error Message Manual, Volume 2* (32650-90152)
- *Compiler Library/XL Reference Manual* (32650-90029)
- *MPE XL System Debug Reference Manual* (32650-90013)
- *Switch Programming User's Guide* (32650-90014)
- *HP Symbolic Debugger/XL User's Guide* (31508-90003)
- *HP TOOLSET/XL Reference Manual* (36044-90001)
- *HP SQL/XL COBOL Application Programming Guide* (36216-90006)

Acknowledgment

At the request of the American National Standards Institute (ANSI), the following acknowledgment is reproduced in its entirety:

> Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgment of the source, but need not quote the acknowledgment):
>
> 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 therewith.
>
> The authors and copyright holders of the copyrighted material used herein 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.
>
> FLOW-MATIC (trademark of 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. F 28-8013, copyrighted 1959 by IBM, FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

Conventions

Notation	Description
	Change bars in the margin show where substantial changes have been made to this manual since the last edition.
`UPPERCASE` and `UNDERLINING`	Within syntax statements, characters in uppercase must be entered in exactly the order shown. Uppercase words that are underlined are keywords that are always required when the clause or statement in which they appear is used in your program. Uppercase words that are *not* underlined are optional, and may be included or omitted. They have no effect on program execution and serve only to make source program listings more readable. The following example illustrates this: `[FILE STATUS IS` *stat-item*`].` `STATUS` must be entered, `FILE` may be either included or omitted. See also "Underlining in dialog" on the following page.
italics	Within syntax statements, a word in italics represents a formal parameter, argument, or literal that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release: `RELEASE` *filename*
punctuation	Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.

`{ }`	Within syntax statements, when several elements within braces are stacked, you must select one. In the following equivalent examples, you select `ON` or `OFF`:
	
	`{ON }`
	`SETMSG {OFF}`
	
	`SETMSG { ON }`
	`{ OFF }`

`{| |}`	Within syntax statements, bars in braces are choice indicators. One or more of the items within the choice indicators must be specified, but a single option may be specified only once.

`[ ]`

Within syntax statements, brackets enclose optional elements. In the following example, brackets around `,TEMP` indicate that the parameter and its delimiter are not required:

```
PURGE filename[,TEMP]
```

When several elements within brackets are stacked, you can select any one of the elements or none. In the following equivalent examples, you can select *devicename* or *deviceclass* or neither:

```
          [devicename]
SHOWDEV   [deviceclass]
```

```
SHOWDEV  [ devicename  ]
         [ deviceclass ]
```

Underlining in dialog

When it is necessary to distinguish user input from computer output, the input is underlined. See also underlining on the previous page.

```
NEW NAME?  ALPHA
```

`[ ] ...`

Brackets followed by a horizontal ellipsis indicate either that a previous bracketed element may be repeated zero or more times, or that elements have been omitted from the description.

```
[WITH DUPLICATES] ...
```

The ellipsis shows that the preceding clause may be repeated indefinitely.

`{ } ...`

Braces followed by a horizontal ellipses indicate either that the item within braces may be repeated one or more times, or that elements have been omitted from the description.

`␣`

Within syntax statements, the space symbol ␣ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank:

```
SET [(modifier)]␣(variable);
```

`<, >, =, <=, >=, <>`

These symbols are used in conditional statements to represent the keywords LESS THAN, GREATER THAN, EQUAL TO, LESS THAN OR EQUAL TO, GREATER THAN OR EQUAL TO, and NOT EQUAL TO, respectively. Although these symbols represent keywords, they are *not* underlined.

;	The semicolon is used only to improve readability and is always optional.
,	The comma is used only to improve readability, and is always optional.
.	The period is a terminator or delimiter that is always required where shown; it must always be entered at the end of every division name, section name, paragraph name, and sentence.
^	The caret is occasionally used in examples to represent an implied decimal point in computer memory.
`Shading`	Features that are part of the 1985 ANSI standard are `shaded`. They are accessible through the ANSI85 entry point.
LG200026_198	In some diagrams and tables, a number appears in the lower left corner. This number is for HP control purposes only and should not be interpreted as part of the diagram or table.

Contents

Figures

Tables

1

Introduction

HP COBOL II/XL is Hewlett-Packard's implementation of the 1985 ANSI COBOL standard (X3.23-1985) and the 1974 ANSI COBOL standard (X3.23-1974), the COBOL programming languages that meet the 1985 and 1974 standards set by the American National Standards Institute (ANSI).

The HP COBOL II/XL compiler compiles COBOL'74 programs as well as COBOL'85 programs. When you invoke it through its ANSI74 entry point (using the COB74XL command file), it accepts only syntax that conforms to COBOL'74. When you invoke it through its ANSI85 entry point (using the COB85XL command file), it accepts the syntax of COBOL'85 plus the intrinsic functions that were defined in 1989 by Addendum 1 of the ANSI COBOL'85 standard. The ANSI85 entry point is the default.

Figure 1-1 shows the relationships between the two entry points of the HP COBOL II/XL compiler and the two revisions of the ANSI standard, COBOL'85 and COBOL'74:

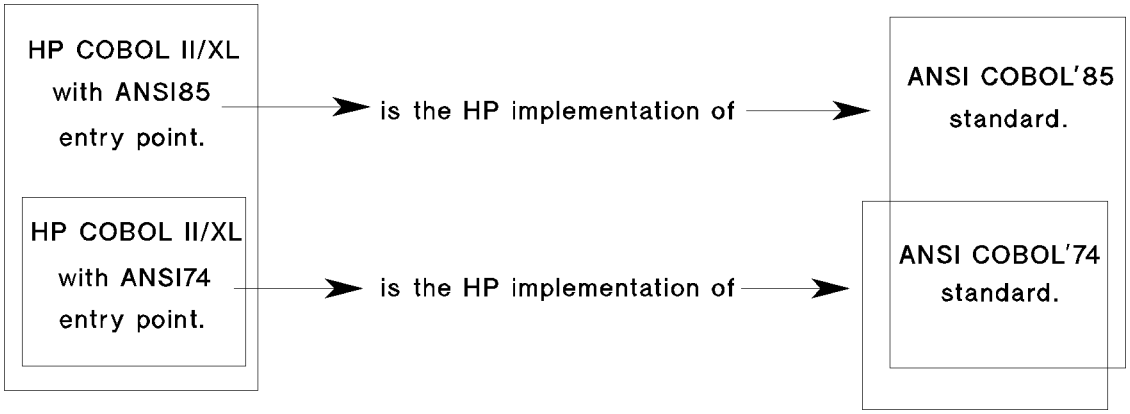


Figure 1-1.
Relationships between HP COBOL II/XL and the ANSI Standards COBOL'74 and COBOL'85

The HP COBOL II/XL product consists of a compiler, which translates HP COBOL II/XL programs into machine object files, and a run-time library. The object code that the compiler generates contains calls to routines in the run-time library and, if intrinsic functions are used, to other languages' run-time libraries.

Note Hereafter in this manual, *intrinsic functions* will be referred to as *COBOL functions*.

Table 1-1 lists components of the HP COBOL II/XL product and describes their use.

Table 1-1. Components of HP COBOL II/XL

File Name	Use
COBOL.PUB.SYS	The HP COBOL II/XL compiler.
in XL.PUB.SYS	The HP COBOL II/XL run-time library.
COBCNTL.PUB.SYS	Source file you can use to override the compiler defaults for the compiler options. The compiler automatically includes a file named COBCNTL.PUB.SYS in each source textfile.
COB85XL.PUB.SYS COB85XLK.PUB.SYS COB85XLG.PUB.SYS COB74XL.PUB.SYS COB74XLK.PUB.SYS COB74XLG.PUB.SYS	Command files you can use to compile, link, and run HP COBOL II/XL programs.
COBCAT.PUB.SYS	Error message catalog file used by the compiler and the run-time library.
COBMAC.PUB.SYS	Macro file used by the run-time library to display error information when a trap is detected.
COBEDIT.PUB.SYS	Program you can use to develop and maintain COPY libraries.

Debugging COBOL Programs

HP COBOL II/XL runs on the MPE XL operating system. You can use the debuggers that run on MPE XL to debug your HP COBOL II/XL programs. They are Debug (the MPE XL System Debugger) and one of two symbolic debuggers: HP Symbolic Debugger/XL or HP TOOLSET/XL.

Subsystems that Interface with HP COBOL II/XL

Table 1-2 lists HP subsystems with which HP COBOL II/XL can interface.

Table 1-2. Subsystems that Interface with HP COBOL II/XL

Subsystem	Description	Where to Look for Details
DEBUG	MPE XL System Debugger.	*System Debug Reference Manual*
HP Symbolic Debugger/XL	A full-featured symbolic debugger that is interactive at the source level.	*HP Symbolic Debugger/XL User's Guide*
HP TOOLSET/XL	A programming environment for developing COBOL programs. It provides source management, a symbolic debugger, and an editor that is specifically for COBOL.	*HP TOOLSET/XL Reference Manual*
TurboIMAGE/XL	A network database management system. Your COBOL program accesses TurboIMAGE/XL routines with intrinsic calls.	*TurboIMAGE/XL Reference Manual*
HPSQL	A relational database management system whose COBOL preprocessor has macros that generate calls to HPSQL.	*HPSQL/XL COBOL Application Programming Guide*
HP System Dictionary/XL	A dictionary of MPE XL data elements.	*HP System Dictionary/XL General Reference Manual*

2

Features of the 1985 ANSI Standard

Introduction

Throughout the rest of this manual, the term *ANSI85* means "HP COBOL II/XL as invoked through its ANSI85 entry point," and the term *ANSI74* means "HP COBOL II/XL as invoked through its ANSI74 entry point."

ANSI85 features fall into these categories:

Category	Description of Category
ANSI85 Features	Features of ANSI COBOL 1985 that ANSI COBOL 1974 does not have.
Post ANSI85 Features	ANSI features implemented since ANSI COBOL 1985. Currently, this consists only of the COBOL functions.
Obsolete Features	Features of ANSI COBOL 1985 that will be deleted from the next full revision of the ANSI COBOL standard.
Incompatible Features	Features of ANSI COBOL 1985 that do not work the same way as the corresponding ANSI COBOL 1974 features.
HP Extensions	Features of HP COBOL II/XL that ANSI COBOL 1974 and 1985 do not include.

This chapter explains ANSI85, obsolete, and incompatible features. For obsolete and incompatible features, it gives justifications for their being in those categories. HP extensions to ANSI COBOL are listed in Chapter 3, "Portability." For a description of the COBOL functions, see the *HP COBOL II/XL Reference Manual*.

ANSI85 Features

ANSI85 features are those that ANSI74 does not have. If you use them in your program, you must invoke the COBOL compiler through its ANSI85 entry point.

The ANSI85 features are listed below, by division. Those marked with an asterisk (*) support structured programming, and are explained in Chapter 3. The others are explained in this chapter, by division.

Division	ANSI85 Feature
Not part of a division	* END PROGRAM header
IDENTIFICATION DIVISION	INITIAL clause
	* COMMON clause
ENVIRONMENT DIVISION	CLASS clause
	SYMBOLIC CHARACTERS clause
DATA DIVISION	EXTERNAL data items and files
	FILLER is now optional
	* GLOBAL data items and files
	USAGE data item formats
PROCEDURE DIVISION	ADD statement enhancement
	ALPHABETIC-LOWER
	ALPHABETIC-UPPER
	CALL BY CONTENT
	* CONTINUE statement
	De-editing
	* EVALUATE statement
	* Explicit scope terminators
	INITIALIZE statement
	INSPECT CONVERTING statement
	* NOT phrases
	* PERFORM statement enhancements
	Reference modification
	Relational operators
	REPLACE statement
	Setting switches
	Setting condition names
	Table Initialization
	* USE GLOBAL AFTER ERROR
	PROCEDURE ON statement

The END PROGRAM header is not considered to be part of any division. It supports nested and concatenated programs and is explained in Chapter 3.

ANSI85 Features in the IDENTIFICATION DIVISION:

This section explains the INITIAL clause, one ANSI85 feature of the IDENTIFICATION DIVISION. The other ANSI85 feature of the IDENTIFICATION DIVISION, the COMMON clause, supports structured programming, and is explained in Chapter 3.

The INITIAL Clause

The INITIAL clause is in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION. It specifies that the program is in an initial state whenever it is called, not only when it is canceled. In an initial state, data is initialized to the values specified in VALUE clauses.

If a program that specifies the INITIAL clause contains other programs (directly or indirectly), the INITIAL clause applies to those programs also.

The INITIAL clause has the same effect as the DYNAMIC control option.

Example

```
PROGRAM-ID.  SUB-PROG INITIAL.
        ⋮
DATA DIVISION.
WORKING-STORAGE SECTION.
01   A-COUNT         PIC 9(8) COMP-3  VALUE ZEROS.
01   B-COUNT         PIC 9(8) COMP-3  VALUE ZEROS
01   CONV-FIELD.
     05  YY   PIC XX.
     05  MM   PIC XX.
     05  DD   PIC XX.
         ⋮
```

The fields A-COUNT and B-COUNT are initialized to zeros each time this subprogram is called, but the initial value of CONV-FIELD is undefined.

The COMMON Clause

The other ANSI85 feature in the IDENTIFICATION DIVISION is the COMMON clause. The COMMON clause supports structured programming and is explained in Chapter 3.

ANSI85 Features in the ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION has two ANSI85 features:

- CLASS clause.
- SYMBOLIC CHARACTERS clause.

Both are in the SPECIAL-NAMES paragraph.

Note The SPECIAL-NAMES paragraph (in the ENVIRONMENT DIVISION) cannot appear in nested programs. All items in the SPECIAL-NAMES paragraph are implicitly global.

CLASS Clause

The CLASS clause is in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. It defines a class for use in a class condition in the PROCEDURE DIVISION. (The CLASS clause allows a user-defined class, in addition to the pre-existing ALPHABETIC and NUMERIC classes.)

Example

The following shows an example of the CLASS clause defining a class `VALID-GRADE`:

```
SPECIAL-NAMES.
    CLASS VALID-GRADE IS "A" "B" "C" "D" "F".
```

The following example shows how the class `VALID-GRADE` could be used:

```
WORKING-STORAGE SECTION.
    01 GRADE-LIST.
       05 CLASS-GRADES PIC X OCCURS 7 TIMES.
                 ⋮
IF GRADE-LIST IS NOT VALID-GRADE THEN PERFORM ERROR-ROUTINE.
```

The above `IF` statement will perform ERROR-ROUTINE if GRADE-LIST contains a character other than A, B, C, D, or F.

SYMBOLIC CHARACTERS Clause

The SYMBOLIC CHARACTERS clause is in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. It equates names with ASCII character numbers, creating figurative constants. You can use it to name and refer to characters whose ASCII values are in the range 1..256. It is especially useful for referencing unprintable characters.

Example

The following shows the SYMBOLIC CHARACTERS clause:

```
SYMBOLIC CHARACTERS BELL IS 8, CARRIAGE-RETURN IS 14.
```

The above statement in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION equates the names BELL and CARRIAGE-RETURN with the unprintable characters for the bell (ASCII character number 8) and carriage return (ASCII character number 14). In your program, you can refer to these characters by the names BELL and CARRIAGE-RETURN, as in the following statement:

```
DISPLAY BELL "JOB COMPLETED" CARRIAGE-RETURN.
```

Note The first character of the COBOL character set is one, not zero. The COBOL characters in the preceding example and their binary, octal, decimal, and hexadecimal representations are:

COBOL Character	Binary Representation	Octal Representation	Decimal Representation	Hexadecimal Representation
8	0111	7	7	7
14	1101	15	13	D

ANSI85 Features in the DATA DIVISION

This section explains the following ANSI85 features of the DATA DIVISION:

- EXTERNAL data items and files.
- The keyword FILLER is now optional.
- The USAGE data item formats BINARY and PACKED-DECIMAL.

The other ANSI85 feature of the DATA DIVISION, the GLOBAL clause, supports structured programming, and is explained in Chapter 3.

EXTERNAL Data Items and Files

EXTERNAL data items and files can be shared by two or more programs. They provide another way to pass information between programs. With the EXTERNAL clause, separately compiled programs can share files (nested programs can share data and files using the GLOBAL clause).

Each program must declare the shared EXTERNAL items that it uses. Shared items are not passed through the USING phrase. The linker matches the EXTERNAL items by name. Therefore, their names must be exactly the same in each program. For more information, see Chapter 4.

Note Data items and files can be declared both EXTERNAL and GLOBAL.

FILLER

The keyword FILLER can be omitted for data items that are never referenced. This saves coding time and makes the code easier to read.

Example

The following shows two ways of specifying a record. One uses FILLER and the other omits it:

```
01  A.
  05  B       PIC X(5).
  05  FILLER  PIC X(5) VALUE "NAME:".

01  A.
  05  B       PIC X(5).
  05          PIC X(5) VALUE "NAME:".
```

USAGE IS BINARY and USAGE IS PACKED-DECIMAL

BINARY and PACKED-DECIMAL usage are alternatives to the default, DISPLAY (one digit per byte). They are the standard for specifying radixes of two (binary) and ten (packed decimal).

In the past, BINARY was expressed as the implementor-defined COMP and PACKED-DECIMAL was defined as the HP extension COMP-3. COMP and COMP-3 still work the same, but BINARY and PACKED-DECIMAL allow greater future portability between machines.

When deciding whether to use DISPLAY, BINARY, or PACKED-DECIMAL for a data item, consider the following:

- How the data item is used:

  Is it used in arithmetic or printed? If it is used in arithmetic, what are the formats of the other operands in the expressions? Avoid mixing formats, which necessitates conversion.

- Storage space:

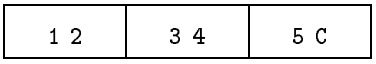
  A data item of the format S9(9) BINARY occupies four bytes. A data item of the format S9(9) PACKED-DECIMAL occupies (*number_of_digits*+1)/2 bytes (rounded up to the nearest whole number). In most cases, BINARY data items occupy less space than PACKED-DECIMAL data items. (See also "Coding Heuristics" in Chapter 3).

Example

The following shows some example fields declared in WORKING-STORAGE:

```
01  VAR-FIELDS.
    05  VAR1       PIC S9(5) PACKED-DECIMAL VALUE +12345.
    05  VAR2       PIC S9(9) BINARY VALUE +12345.
```

In the above example, `VAR1` is stored in three bytes because PACKED-DECIMAL allows byte-alignment and allocates only the number of bytes required for the defined field. The following shows how `VAR1` might be stored in memory:



VAR2 is stored in four bytes. Binary fields are stored in the two's complement form, requiring two, four, or eight bytes each. ∎

ANSI85 Features in the PROCEDURE DIVISION

This section explains the following ANSI85 features of the PROCEDURE DIVISION:

- ADD statement enhancement.
- ALPHABETIC-LOWER.
- ALPHABETIC-UPPER.
- CALL BY CONTENT.
- De-editing.
- INITIALIZE statement.
- INSPECT CONVERTING statement.
- Reference modification.
- Relational operators.
- REPLACE statement.
- Setting switches.
- Setting condition names.
- Table initialization.

The other ANSI85 features of the PROCEDURE DIVISION support structured programming, and are explained in Chapter 3. These are:

- CONTINUE statement.
- EVALUATE statemnet.
- Explicit scope terminators.
- NOT phrases.
- PERFORM statement enhancements.
- USE GLOBAL AFTER ERROR PROCEDURE ON statement.

ADD Statement Enhancement

An ADD statement in ANSI85 can have both a TO phrase and a GIVING phrase. All literals and values of the identifiers to the left of the GIVING keyword are added and the result is stored into each identifier named to the right of the GIVING keyword. See Format 2 of the ADD statement in the *HP COBOL II/XL Reference Manual*.

Example

The following two statements are equivalent:

```
ADD A TO B GIVING C
ADD A B GIVING C
```

ALPHABETIC-LOWER and ALPHABETIC-UPPER Class Tests

The class test ALPHABETIC-LOWER returns TRUE if every character of the specified data item is a lowercase letter or a space. The class test ALPHABETIC-UPPER returns the value TRUE if every character of a specified data item is an uppercase letter or a space.

Example

The following show two IF statements that use the ALPHABETIC-LOWER and ALPHABETIC-UPPER class conditions:

```
IF STRING1 IS ALPHABETIC-LOWER PERFORM UPSHIFT.
IF STRING2 IS ALPHABETIC-UPPER THEN PERFORM CAPITAL.
```

CALL BY CONTENT

When your program passes an actual parameter BY CONTENT, it copies the actual parameter and passes the address of the copy to the subprogram. If the subprogram changes the value of its formal parameter, it changes the value of the copy, but it does not change the value of your program's actual parameter. For more information on parameter passing, see Chapter 4.

CALL BY CONTENT has a performance penalty, because each parameter passed BY CONTENT must be copied.

De-Editing

De-editing converts an edited numeric field to its numeric value, allowing you to move it to either a numeric field or a numeric edited field.

Example

The following shows an example of a de-edited move:

```
WORKING-STORAGE SECTION.
  01   PRINT-A       PIC $ZZZ,ZZZ.99CR.
  01   HOLD-A        PIC S9(6)V99.
PROCEDURE DIVISION.
PARA-001.
    MOVE -76543.21 TO PRINT-A.
    MOVE PRINT-A TO HOLD-A.
```

A de-edited MOVE statement.

The first move statement above sends the following data to PRINT-A:

```
-076543.21
```

- The following value is stored in PRINT-A:

```
$ 76,543.21CR
```

The second move statement is the de-edited move. It sends the following data to HOLD-A:

```
$ 76,543.21CR
```

- The following value is stored in HOLD-A. (There is an implied decimal point between 2 and 3):

```
0765432J
```

All edit symbols are removed and blanks are converted to zeros when the edited value is moved.

INITIALIZE Statement

The INITIALIZE statement sets the values of specified types of elementary items in a record to specified values.

Example

The following example shows the INTIALIZE statement:

```
WORKING-STORAGE SECTION.
01  RECORD-1.
    05  EMP-NO    PIC 9(6).
    05  EMP-NAME  PIC X(20).
    05  EMP-PAY   PIC 9(5)V99.
    05  JOB-TITLE PIC X(20).
                 ⋮
PROCEDURE DIVISION.
MAIN-100.
   INITIALIZE RECORD-1 REPLACING NUMERIC BY ZERO
                       REPLACING ALPHANUMERIC BY SPACES.
```

The above INITIALIZE statement has the same effect and efficiency as the following MOVE statements:

```
    MOVE ZERO TO EMP-NO EMP-PAY.
    MOVE SPACES TO EMP-NAME JOB-TITLE.
```

Note that if the record to be initialized contains only elementary items with fillers or items of the wrong category, the INITIALIZE statement has no effect. An error message is output.

INSPECT CONVERTING Statement

The INSPECT CONVERTING statement is similar to the INSPECT REPLACING statement, but it is more efficient. It allows you to specify several replacements in one string, rather than requiring an entire line for each replacement.

Example 1

The following two INSPECT statements are equivalent:

```
INSPECT WORD CONVERTING "ABCD" TO "XYZX" AFTER QUOTE BEFORE "#".

INSPECT WORD REPLACING
      ALL "A" BY "X" AFTER QUOTE BEFORE "#"
      ALL "B" BY "Y" AFTER QUOTE BEFORE "#"
      ALL "C" BY "Z" AFTER QUOTE BEFORE "#"
      ALL "D" BY "X" AFTER QUOTE BEFORE "#".
```

If in the above example the initial value of WORD is:

```
AC"AEBDFBCD#AB"D
```

Then the final value of WORD is:

```
AC"XEYXFYZX#AB"D
```

Converting uppercase letters to their lowercase forms is much easier with the INSPECT CONVERTING statement than it would be with the INSPECT REPLACING statement.

Example 2

The following two INSPECT statements are equivalent:

```
INSPECT NAME CONVERTING
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO "abcdefghijklmnopqrstuvwxyz".

INSPECT NAME REPLACING
      ALL "A" TO "a"
      ALL "B" TO "b"
            :
      ALL "Z" TO "z".
```

Example 3

The following INSPECT CONVERTING statement translates blanks and asterisks to zeros:

```
INSPECT AMT-DUE CONVERTING " *" TO "00"
```

Reference Modification

Reference modification allows you to reference part of an item whose usage is DISPLAY. To access a substring within a data item, specify the position of the leftmost character and length of the substring, in characters. You can specify the position and length with any integer expression.

Example 1

If the value of the data item A is "ABCDEFGHI", then the following statement moves the value "CDEFG" to the data item B:

```
MOVE A (3:5) TO B
```

Example 2

The data item in a reference modification can also be the target of a move. If the value of the data item A is "ABCDEFGHI", then the following statement gives A the value "AB*****HI".

```
MOVE ALL '*' TO A(3:5)
```

Example 3

This example shows reference modification on the result of a COBOL function call. The example calls the COBOL function CURRENT-DATE and displays only the characters in positions 1 through 4. These characters represent the current year.

```
DISPLAY FUNCTION CURRENT-DATE (1:4).
```

The above DISPLAY statement displays the following:

```
1991
```

See "Reference Modification" in the *HP COBOL II/XL Reference Manual* for more examples.

Relational Operators

In ANSI85 you can use the relational operators LESS THAN OR EQUAL TO (<=) and GREATER THAN OR EQUAL TO (>=). An HP extension allows the symbol <> as shorthand for NOT EQUAL.

Example

The following IF statements use these relational operators:

```
IF TR-CODE <= 1 PERFORM 310-GET-NEXT-RECORD.
IF STATE CODE >= 50 THEN PERFORM FOREIGN-RTN.
IF CITY-CODE <> 25 PERFORM 420-VALIDATE-CITY.
```

REPLACE Statement

The REPLACE statement affects source program text the way the COPY REPLACING statement affects library text. The scope of the REPLACE statement is from its start to the start of another REPLACE statement or the end of the current concatenated program, whichever comes first.

The program in the following example replaces ANSI85 reserved words that were not reserved in the 1974 ANSI standard. Remember that the REPLACE statement is executed each time the program is compiled. It may be more efficient to use an editor to change the file permanently than to consume CPU time to change the file each time it is compiled.

Example

The following shows a COBOL program before REPLACE execution:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    PROG1.
DATA DIVISION.
REPLACE   ==TEST==  BY  ==TESTT==          Begin REPLACE statement 1.
          ==TRUE==  BY  ==TRUE-FLAG==.     End REPLACE statement 1.
   01  NAME   PIC   X(30).
   01  TEST   PIC   X.                     TEST will be replaced.
        88   TRUE    VALUE  "T".           TRUE will be replaced.
PROCEDURE DIVISION.
P1.
     ACCEPT TEST.                          TEST will be replaced.
     IF TRUE   PERFORM P2.                 TRUE will be replaced.
     REPLACE ==ALPHABETIC==                Begin REPLACE statement 2.
        BY  ==ALPHABETIC-UPPER==.          End REPLACE statement 2.
     IF NAME IS ALPHABETIC THEN            ALPHABETIC will be replaced.
        SET TRUE-FLAG TO TRUE.
     REPLACE OFF.                          REPLACE statement 3.
     PERFORM P3 WITH TEST AFTER
        UNTIL NAME IS NOT ALPHABETIC.
              ⋮
```

The actual code sent to the compiler becomes the following:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   PROG1.
DATA DIVISION.
   01  NAME   PIC  X(30).
   01  TESTT  PIC  X.
       88  TRUE-FLAG  VALUE "T".
PROCEDURE DIVISION.
P1.
     ACCEPT TESTT.
     IF TRUE-FLAG PERFORM P2.
     IF NAME IS ALPHABETIC-UPPER THEN
        SET TRUE-FLAG TO TRUE.
     PERFORM P3 WITH TEST AFTER
        UNTIL NAME IS NOT ALPHABETIC.
               ⋮
```

Statement 2 overrides statement 1 and the second occurrence of "TEST" remains unchanged. Statement 3 ends all replacing and the second occurrence of "ALPHABETIC" remains unchanged.

Setting Switches

The SET statement in COBOL can set external switches to the values ON and OFF. An ANSI74 program can test the values of switches, but it cannot change their values.

Example

The following declares a switch:

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
   SW0 IS SWITCH-1
```

The following SET statement uses the switch:

```
PROCEDURE DIVISION.
PRINT-ROUTINE.
   SET SWITCH-1 TO ON.
```

Setting Condition Names

The SET statement in COBOL can set condition names to the value TRUE.

Example

The following declares a condition name, EOF-FLAG:

```
01  READ-FLAG      PIC 9.
    88  EOF-FLAG   VALUE 1.
```

The following SET statement uses the condition name EOF-FLAG:

```
SET EOF-FLAG TO TRUE.
```

The SET statement above is equivalent to the following MOVE statement:

```
MOVE 1 TO READ-FLAG.
```

Example

You cannot set a condition name to FALSE, but you can define two condition names, one for the true case and one for the false case. The following example illustrates this:

```
01   FIRST-TIME-FLAG   PIC X  VALUE "Y".
     88  FIRST-TIME     VALUE "Y".
     88  FIRST-TIME-OFF VALUE "N".
```

The following example uses the SET statement on both condition names:

```
IF FIRST-TIME
   PERFORM INIT-SECTION
   SET FIRST-TIME-OFF TO TRUE
END-IF
```

Table Initialization

You can initialize a table (a data item that contains an OCCURS clause) by specifying a VALUE clause for it. Each table element (or "occurrence") receives the value that you specify.

Example

In the following example, the ten elements of table B receive the value zero:

```
01 A.
   05 B PIC S999 OCCURS 10 TIMES VALUE 0.
```

- For dynamic subprograms, this initialization is as efficient as the same initialization accomplished by a PERFORM loop for every VALUE clause.

Obsolete Features

Obsolete features of ANSI85 are those that will be deleted from the next full revision of the COBOL standard. HP COBOL II/XL supports them, but its successor may not. If you use obsolete features in your program, you may not be able to compile it on compilers that implement the next ANSI COBOL.

This table lists the obsolete features, justifies their being obsolete, and tells you how to make your program independent of them.

Obsolete Elements of ANSI85 COBOL

Obsolete Feature	Justification for Obsolescence	How to Make Your Program Independent of the Feature
AUTHOR paragraph, INSTALLATION paragraph, DATE-WRITTEN paragraph, DATE-COMPILED paragraph, SECURITY paragraph	These paragraphs do not affect program operation. Comments can serve the same purpose as they do. Interaction between the COPY statement and the comment entries in these paragraphs is often ambiguous. For example, how can you tell if a comment entry contains the word COPY or a COPY statement? DATE-COMPILED and SECURITY are implementor-defined comment entry paragraphs, most of which were obsoleted in order to clean up and regularize COBOL.	Either leave it out entirely, or identify the information such as author and installation, with comments in the IDENTIFICATION DIVISION.

Obsolete Elements of ANSI85 COBOL (continued)

Obsolete Feature	Justification for Obsolescence	How to Make Your Program Independent of the Feature
MEMORY-SIZE clause	This feature is a carry-over from the time when many systems required a specification of memory size allocation to load the run unit. Memory capacity for a family of main frame models often ranged from 8K to 64K. COBOL programs used the MEMORY-SIZE clause to generate objects for specific models. In today's computing environment, this function is more appropriately controlled by the host operating system. In 1974 Standard COBOL, the MEMORY-SIZE clause was optional; therefore, no standard conforming COBOL implementations require it.	Leave it out. The operating system performs its function.
MULTIPLE FILE TAPE clause	Allowing users to sequentially access multiple files on a labeled tape without rewinding the tape is a function of the operating system, not the COBOL program.	Instead of using the MULTIPLE FILE TAPE clause or the VALUE OF clause (also obsolete), use a file equation (see Chapter 5 for details).
RERUN clause	Seven forms of the RERUN statement are provided. The RERUN clause provides only half of a complete rerun/restart facility. That is, the syntax and semantics for restart are not specified. Due to the variety of forms of the RERUN clause, there is no guarantee that a program that uses it would be transportable. In today's computing environment, this feature is more appropriately controlled by the host operating system.	Leave it out. The operating system performs its function.

Obsolete Elements of ANSI85 COBOL (continued)

Obsolete Feature	Justification for Obsolescence	How to Make Your Program Independent of the Feature
DATA RECORDS clause	The DATA RECORDS clause gives redundant information and could mislead someone who reads the program.	Leave it out. The same information is in the record description associated with the file.
LABEL RECORDS clause	Specifying the presence of file labels is a function of the operating system, not the COBOL program.	Leave it out. The operating system performs its function.
VALUE OF clause	Describing file label items is a function of the operating system, not the COBOL program.	Leave it out. The operating system performs its function.
REVERSED phrase	The hardware necessary for this function is not widely available; thus, it is infrequently implemented and not appropriate for standardization.	Do not use this phrase. It has never been implemented for HP COBOL II/XL.
ENTER statement	The ENTER statement is optional and implementation defined; therefore, it is not portable and is not appropriate for standardization.	Do not use this phrase. It has never been implemented for HP COBOL II/XL.
STOP LITERAL statement	The STOP LITERAL statement is implementation defined; therefore, it is not portable and is not appropriate for standardization.	Do not use the STOP LITERAL statement. If necessary use DISPLAY to send messages to the console. (If a tape mount is necessary, the operating system will handle the delay.)
ALTER Statement	The ALTER statement makes a program difficult to understand and maintain. The ALTER statement provides no unique function, because the GO TO DEPENDING ON statement can serve the same purpose.	Use the GOTO DEPENDING ON statement instead.

Obsolete Elements of ANSI85 COBOL (continued)

Obsolete Feature	Justification for Obsolescence	How to Make Your Program Independent of the Feature
Debug Module features: * Object time switch (PARM=1 in RUN command), USE FOR DEBUGGING statement, Special register DEBUG-ITEM	Today's computing environment usually provides interactive debug facilities, which provide the function of the Debug Module without requiring COBOL source statements.	Do not use features that support the Debug Module. Refer to the *HP COBOL II/XL Reference Manual*. Debug your program with HP Symbolic Debugger/XL, HP TOOLSET/XL, or DEBUG (the MPE XL System Debugger).

* The following Debug Module features are not obsolete and are now part of the nucleus module:

- WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph.
- Debugging lines (lines with the letter D in column seven).

Incompatible Features

Incompatible features are those that work differently in ANSI85 and ANSI74. When you invoke the COBOL compiler through its ANSI85 entry point, it compiles these features according to ANSI COBOL 1985. When you invoke the compiler through its ANSI74 entry point, it compiles them according to ANSI COBOL 1974.

There are four exceptions, features that both entry points compile according to ANSI COBOL 1985. See "CANCEL and STOP RUN Statements," "EXIT PROGRAM Statement," and "Exponentiation."

ALPHABET Keyword

The keyword ALPHABET is required in the ALPHABET clause of an ANSI85 program. In an ANSI74 program, it is optional.

Example

The following paragraph is legal in ANSI74:

```
SPECIAL-NAMES.  WORD-1 IS WORD-2.
```

In ANSI85, the above must be changed to:

```
SPECIAL-NAMES.  ALPHABET   WORD-1 IS WORD-2.
```

Justification for Changing the ALPHABET Keyword

Implementor-names are system-names. Alphabet-names and mnemonic-names are user-defined words. In ANSI COBOL 1985, system-names and user-defined words form intersecting sets and can therefore contain the same words. In the legal following clause, if WORD-1 is both an implementor-name and an alphabet-name, and WORD-2 is both a mnemonic-name and an implementor-name, then it is impossible to tell whether the implementor-name clause or the alphabet-name clause is intended.

```
SPECIAL-NAMES.
   WORD-1 IS WORD-2.
```

The introduction of the keyword ALPHABET in the alphabet-name clause resolves this ambiguity.

This problem did not exist in ANSI COBOL 1974 because system-names and user-defined words formed disjoint sets (therefore, the above clause was illegal). The keyword ALPHABET did not appear in the alphabet-name clause of the SPECIAL-NAMES paragraph in ANSI COBOL 1974.

Allowing system-names and user-defined words to intersect makes it easier to move a program from implementation to implementation, because system-names need not be changed. To modify an existing program, insert the keyword ALPHABET into the alphabet-name clause.

CANCEL and STOP RUN Statements

Both ANSI85 and ANSI74 entry points conform to ANSI COBOL 1985, which specifies that the CANCEL and STOP RUN statements close all open files. ANSI COBOL 1974 does not specify the status of files that are in open mode when the program is canceled.

Justification for Changing CANCEL and STOP RUN

In 1974 Standard COBOL, the status of files left in the open mode when the program was canceled was not defined. The change in 1985 Standard COBOL produces a predictable result for the CANCEL statement. The only programs that may be affected are those that cancel other programs and expect files associated with the canceled programs to remain open after the CANCEL statements execute.

In 1974 Standard COBOL, the status of files left in the open mode when the program finished executing was not defined. In some cases, this situation could have caused errors. The change in the 1985 Standard COBOL produces a predictable result for the STOP RUN statement. Few programs will be affected, because many implementations already close files after executing the STOP RUN statement.

EXIT PROGRAM Statement

Both ANSI85 and ANSI74 entry points conform to ANSI COBOL 1985, which specifies that an implicit EXIT PROGRAM statement is executed when there is no next executable statement in a called program. ANSI COBOL 1974 does not specify the action in this situation.

If you want to detect an error in this situation, instead of executing an implicit EXIT PROGRAM statement, end your program like this:

```
999-END-PROG SECTION.
    DISPLAY "999-END-PROG THIS SHOULD NEVER PRINT".
```

Justification for Changing EXIT PROGRAM

In 1974 Standard COBOL, this situation was undefined. The change in 1985 Standard COBOL makes programs more transportable. The change only affects programs that depend on another implementation action when the EXIT PROGRAM statement is omitted.

Exponentiation

Both ANSI85 and ANSI74 entry points conform to ANSI COBOL 1985, which specifies the following:

- If an expression whose value is zero is raised to a negative or zero power, a size error occurs.
- If the value of an exponentiation is not a real number, a size error occurs.

ANSI COBOL 1974 did not address these special cases of exponentiation.

Example

The following expressions cause size error conditions because they raise the value zero to negative or zero powers:

```
0**0
0**(-2)
((4*3)-(2*6))**(5-7)
```

The following expression causes a size error condition because it takes the square root of a negative number:

```
-2**(1/2)
```

Justification for Changing Exponentiation

1974 Standard COBOL did not state what would happen in these special cases of exponentiation, so implementors were free to decide how to handle them. The change in 1985 Standard COBOL resolves an undefined situation and promotes program portability. The change affects few programs, because two of the previously undefined cases caused errors and the other case is consistent with most implementations.

OCCURS Clause

When a receiving item contains an OCCURS clause with a DEPENDING ON phrase, and the receiving item also contains the object of the DEPENDING ON phrase, ANSI85 assumes that the object has its maximum length.

In the same situation, ANSI74 assumes that the object has the length of its current value. Consequently, a MOVE or READ INTO statement can result in loss of data unless you change the value of the object of the DEPENDING ON phrase before you change the value of the entire receiving item.

Example

The following example shows two records that contain tables with OCCURS DEPENDING ON clauses, where the OCCURS DEPENDING ON items, `A-SIZE` and `B-SIZE`, are part of the record:

```
FD  INPUT-FILE.
01  A.
    02  A-TABLE.
        03  A-SIZE  PIC 99.
        03  A-ITEM  OCCURS 1 TO 10 TIMES DEPENDING ON A-SIZE.

WORKING-STORAGE SECTION.
01  B.
    02  B-TABLE.
        03  B-SIZE  PIC 99.
        03  B-ITEM  OCCURS 1 TO 10 TIMES DEPENDING ON B-SIZE.
```

In the preceding program fragment, assume that the value of `A-SIZE` is 10 and the value of `B-SIZE` is five. In ANSI74, the following statements move all of the data in A to B:

```
MOVE A-SIZE TO B-SIZE.
MOVE A TO B.
```

In ANSI85, the following statement moves all of the data in A to B:

```
MOVE A TO B.
```

In ANSI74, the following statements read INPUT-FILE into B:

```
READ INPUT-FILE.
MOVE A-SIZE TO B-SIZE.
MOVE A TO B.
```

In ANSI85, the following statement reads INPUT-FILE into B:

```
READ INPUT-FILE INTO B.
```

Justification for Changing the OCCURS Clause

1974 Standard COBOL computed the value of the length based on the value of the item in the DEPENDING ON phrase prior to execution of the statement. Using 1974 Standard COBOL rules with a MOVE or READ INTO statement could have caused loss of data if the value of the DEPENDING ON data item was not set to indicate the length of the sending data before the MOVE or READ INTO statement executed.

This change does not affect programs that conform to 1974 Standard COBOL. To change an affected program, restructure the affected data records so that data items do not follow variable-length items.

READ NEXT after OPEN I-O, WRITE and REWRITE Statements

In ANSI85, for a relative or indexed file in dynamic access mode, a READ NEXT statement that follows an OPEN I-O statement and one or more WRITE statements accesses the first record in the file when the READ NEXT statement is executed.

In ANSI74, for a relative or indexed file in dynamic access mode, a READ NEXT statement that follows an OPEN I-O statement and one or more WRITE statements accesses the first record in the file when the OPEN I-O statement is executed. Therefore, if one of the WRITE statements inserts a record before the original first record, the READ NEXT statement accesses the original first record instead of the new first record.

Example

The file looks like the following when the OPEN I-O statement is executed:

Original First Record (key=*n*)	Record	Record	...

A WRITE statement inserts a new first record so that the file looks like the following:

In ANSI74, a READ NEXT statement accesses this record.

↓

New First Record (key=*n*-3)	Original First Record (key=*n*)	Record	Record	...

↑

In ANSI85, a READ NEXT statement accesses this record.

Justification for Changing READ NEXT

It is considered more logical that on execution of the first READ statement after an OPEN statement, the record accessed is the first record in the file at the time that the READ statement is executed.

VARYING ... AFTER Phrase in PERFORM Statement

In the VARYING ... AFTER phrase in a PERFORM statement, ANSI85 augments *identifier-2* before it sets *identifier-5*. ANSI74 performs these steps in reverse order: it sets *identifier-5* before it augments *identifier-2*.

The reason for this change is that the ANSI74 PERFORM statement was often misinterpreted, resulting in incorrect programs.

ANSI85 and ANSI74 produce different results when *identifier-5* depends on *identifier-2* or vice versa.

Example

In the following PERFORM statement, Y depends on X:

```
PERFORM PARA3 VARYING X FROM 1 BY 1 UNTIL X IS GREATER THAN 3
    AFTER Y FROM X BY 1 UNTIL Y IS GREATER THAN 3
```

ANSI85 executes PARA3 six times with the following values for X and Y:

```
X:  1  1  1  2  2  3
Y:  1  2  3  2  3  3
```

ANSI74 executes PARA3 eight times with the following values for X and Y:

```
X:  1  1  1  2  2  2  3  3
Y:  1  2  3  1  2  3  2  3
```

In ANSI85 (but not ANSI74), the following statement sequence is equivalent to the statement above:

```
     PERFORM PARA2 VARYING X FROM 1 BY 1 UNTIL X IS GREATER THAN 3.

PARA2.
    PERFORM PARA3 VARYING Y FROM X BY 1 UNTIL Y IS GREATER THAN 3.
```

In ANSI85 (but not ANSI74), the above statement sequence is equivalent to the following nested PERFORM statement:

```
PERFORM VARYING X FROM 1 BY 1 UNTIL X IS GREATER THAN 3
    PERFORM VARYING Y FROM X BY 1 UNTIL Y IS GREATER THAN 3
        code for PARA-3
    END-PERFORM
END-PERFORM.
```

Justification for Changing PERFORM VARYING

The situation where one VARYING variable depends on another is useful for processing half a matrix along the diagonal. The rules of 1985 Standard COBOL specify this function properly, while the rules of 1974 Standard COBOL did not. This change affects few existing programs.

File Status Codes

File status code incompatibilities between ANSI85 and ANSI74 file status codes are shown in Table 5-9.

Justification for Changing File Status Codes

1974 Standard COBOL specified only a few file status code conditions. This made the following true:

- A COBOL program could not distinguish the many different exceptional conditions and treat them differently.
- Each implementor specified a different set of implementor-defined status codes to cover many situations in many ways.
- The results of many I-O situations were undefined. That is, 1974 Standard COBOL stated that certain criteria were to be met, but not what happened when they were not met.

1985 Standard COBOL attempts to define file status codes for these undefined I-O situations, so that a COBOL program can check for these error conditions in a standard way and take corrective action where appropriate.

The addition of new file status codes affects the following kinds of programs:

- Programs that check specific file status code values.
- Programs that test for specific implementor-defined status values to detect conditions that are now defined.
- Programs that rely on a successful completion status for any of the conditions that are now defined (in the case of new file status values 04, 05, and 07, this only affects programs that examine both character positions of the file status to check for successful completion).
- Programs that rely on an implementor-dependent action (such as abnormal termination of the program) when a newly defined condition arises.

The STAT74 control option causes the compiler to follow ANSI74 rules even when it is invoked through its ANSI85 entry point. See Chapter 6 for more information on STAT74.

3

Programming Practices

Introduction

This chapter describes programming practices that can help you do the following:

- Make your programs structured, so that it is easier to design, code, read, and maintain.
- Make your programs faster at run-time.
- Make your programs more portable, so that you can run it on other machines with minimal changes.

Structured Programming

Structured programming makes your program easier to design, code, read, and maintain. It is loosely defined as programming that stresses clear top-down design:

- A complex problem is broken into functionally cohesive modules that perform simple tasks.
- The structure and control flow of each module reflect the programmer's approach to the problem.
- Control flows from top to bottom (it is not transferred from one module to the middle of another module).

The COBOL'85 features that support and encourage structured programming are listed below, by division. These are ANSI85 features. COBOL74 does not have them. If you use them in your program, you must invoke the HP COBOL II/XL compiler through its ANSI85 entry point.

Division	**ANSI85 Structured Programming Feature**
Not part of a division	END PROGRAM header
IDENTIFICATION DIVISION	COMMON clause
DATA DIVISION	GLOBAL data items and files
PROCEDURE DIVISION	CONTINUE statement
	EVALUATE statement
	Explicit scope terminators
	NOT phrases
	PERFORM statement enhancements
	USE GLOBAL AFTER ERROR PROCEDURE ON statement

This section explains the following:

- ANSI85 structured programming features, by division.
- When to use nested programs and GLOBAL data.

Note The SPECIAL-NAMES paragraph (in the ENVIRONMENT DIVISION) cannot appear in nested programs. All items in the SPECIAL-NAMES paragraph are implicitly GLOBAL.

END PROGRAM Header

The END PROGRAM header ends a COBOL source program explicitly, whereas the absence of additional source lines ends a program implicitly. When a single source file contains more than one COBOL program, all but the last unnested program must end explicitly, with an END PROGRAM header.

When COBOL programs are nested, their hierarchy is described by the sequence of PROGRAM-ID paragraph/END PROGRAM header pairs. A PROGRAM-ID paragraph and END PROGRAM header are a pair if they specify the same program name.

Separately compiled programs within a run unit must have unique names. Within a single separately compiled program, nested programs must have unique names. (Nested programs are not considered to be separately compiled.)

Note The HP extension ID DIVISION is not supported for applications that use nested or concatenated programs. Use IDENTIFICATION DIVISION instead.

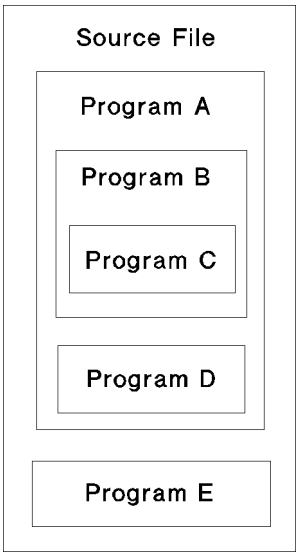
Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. A.
PROCEDURE DIVISION.
BEGIN-A.
DISPLAY "A IS THE OUTERMOST PROGRAM".
CALL "B".
CALL "D".
CALL "E".

IDENTIFICATION DIVISION.
PROGRAM-ID. B.
PROCEDURE DIVISION.
BEGIN-B.
DISPLAY "B IS NESTED WITHIN A".
CALL "C".

IDENTIFICATION DIVISION.
PROGRAM-ID. C.
PROCEDURE DIVISION.
BEGIN-C.
DISPLAY "C IS NESTED WITHIN B".
END PROGRAM C.

END PROGRAM B.

IDENTIFICATION DIVISION.
PROGRAM-ID. D.
PROCEDURE DIVISION.
BEGIN-D.
DISPLAY "D IS NESTED WITHIN A".
END PROGRAM D.

END PROGRAM A.

IDENTIFICATION DIVISION.
PROGRAM-ID. E.
PROCEDURE DIVISION.
BEGIN-E.
DISPLAY "E IS A CONCATENATED PROGRAM".
```

In the preceding example, the PROGRAM-ID paragraph/END PROGRAM header pairs describe this nesting hierarchy:

- Program A is the parent of programs B and D. (B and D are siblings).
- Program A is the grandparent of program C. (C is a child of program B.)
- Program E is a concatenated program. That is, it is in the same source file as A, B, C, and D, but is not nested within any of them. It is considered to be a separately compiled program. (It does not need an END PROGRAM header, because it is the last unnested program in the source file.)

The relationship between these programs is shown graphically below:



By default, a program can only call its children and separately compiled programs. It cannot call its siblings, its grandchildren, or their descendants. (In the preceding example, program A can call B, D, and E, but not C.) The COMMON clause allows exceptions to this rule.

IDENTIFICATION DIVISION: COMMON Clause

The COMMON clause allows a nested program to be called by its siblings and their descendants, as well as its parent. (By default, it can only be called by its parent.) The COMMON clause does not allow recursion; that is, the common program cannot be called by its descendants or itself.

See "Call Rules" in Chapter 4 for an example of the COMMON clause.

DATA DIVISION: GLOBAL Data Items and Files

GLOBAL data items and files can be accessed from any program nested within the program that declares it (that is, all of its descendants).

The GLOBAL clause can appear only on 01 level data items, or on an FD level indicator. Items subordinate to a GLOBAL data item or file are also GLOBAL. GLOBAL items cannot be declared in the LINKAGE SECTION.

Example

```
01 GLOBAL-DATA IS GLOBAL.
   05  ITEM1                PICTURE XX.
   05  ITEM2                PICTURE 99.
```

When one program contains another program, the two programs can use the same data item names (in different DATA DIVISIONs). Unless all of these items are EXTERNAL, they refer to distinct data items. When the compiler encounters such a name in the PROCEDURE DIVISION, it assumes that it applies to the first data item it finds that meets the qualifications.

For example, in the following program, a GLOBAL data item, ITEM-A, is declared and displayed in the program OUTER. The program OUTER directly contains the program NESTED-1. When NESTED-1 displays ITEM-A, it is referencing the ITEM-A declared in OUTER.

However, NESTED-2, a program directly contained by NESTED-1, declares another ITEM-A. When NESTED-2 displays ITEM-A, it is referencing the most local ITEM-A (the one it has declared itself). (If NESTED-2 contained nested programs, it might be desirable for it to declare its own ITEM-A to be GLOBAL, so that it would be available to its nested programs.)

The compiler first tries to resolve the reference to a data item in the current program. If it does not find it, it searches for GLOBAL items in the enclosing program(s).

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OUTER.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-A  IS GLOBAL     PICTURE X(20) VALUE "Global item in OUTER".
PROCEDURE DIVISION.
BEGIN.
DISPLAY ITEM-A.
CALL "NESTED-1".

IDENTIFICATION DIVISION.
PROGRAM-ID. NESTED-1.
PROCEDURE DIVISION.
BEGIN.
DISPLAY ITEM-A.
CALL "NESTED-2".

IDENTIFICATION DIVISION.
PROGRAM-ID. NESTED-2.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ITEM-A                PICTURE X(23) VALUE "Local item in NESTED-2".
PROCEDURE DIVISION.
BEGIN.
DISPLAY ITEM-A.
END PROGRAM NESTED-2.

END PROGRAM NESTED-1.

END PROGRAM OUTER.
```

The above program displays the following:

```
Global item in OUTER.
Global item in OUTER.
Local item in NESTED-2
```

A data item or file can be declared both GLOBAL and EXTERNAL, in which case it is accessible to all the nested programs with a single declaration and to all the separately compiled programs that declare it EXTERNAL.

PROCEDURE DIVISION

The PROCEDURE DIVISION has the following ANSI85 structured programming features:

- CONTINUE statement.
- EVALUATE statement.
- Explicit scope terminators.
- NOT phrases.
- PERFORM statement enhancements.
- USE GLOBAL AFTER ERROR PROCEDURE ON statement.

CONTINUE Statement

The CONTINUE statement is a nonexecutable substitute for a conditional or imperative statement or for the keyword EXIT in an EXIT paragraph.

Example.

```
IF A < B THEN
   IF A < C THEN
      CONTINUE
   ELSE
      MOVE ZERO TO A
   END-IF
   ADD B TO C.
SUBTRACT C FROM D.
```

The CONTINUE statement allows control to go to the ADD statement if A is less than C.

EVALUATE Statement

The EVALUATE statement is a multicondition, multibranch statement. It evaluates sets of conditions. The first time all the conditions in a set are true, it executes the associated group of statements. (Each condition arises from the comparison of a subject with an object. Refer to the *HP COBOL II/XL Reference Manual* for details.)

Example 1.

```
EVALUATE HOURS-WORKED ALSO EXEMPT
   WHEN 0           ALSO ANY  PERFORM NO-PAY
   WHEN 1  THRU 40  ALSO ANY  PERFORM REG-PAY
   WHEN 41 THRU 80  ALSO "N"  PERFORM OVERTIME-PAY
   WHEN 41 THRU 80  ALSO "Y"  PERFORM REG-PAY
   WHEN OTHER                 PERFORM PAY-ERROR
END EVALUATE
```

The sets of conditions in the above EVALUATE statement are:

1. HOURS-WORKED is 0 and EXEMPT is any value.
2. HOURS-WORKED is a number from 1 through 40 and EXEMPT is any value.
3. HOURS-WORKED is a number from 41 through 80 and EXEMPT contains “N”.
4. HOURS-WORKED is a number from 41 through 80 and EXEMPT contains “Y”.

If condition 1 is true, NO-PAY is performed. If condition 1 is false and condition 2 is true, REG-PAY is performed. If conditions 1 and 2 are false and condition 3 is true, OVERTIME-PAY is performed. If conditions 1, 2, and 3 are false and condition 4 is true, REG-PAY is performed. If conditions 1, 2, 3, and 4 are false, PAY-ERROR is performed.

You can always write an EVALUATE statement that is equivalent to a nested IF statement, but you cannot always write a nested IF statement that is equivalent to an EVALUATE statement. This is because there is a limit to the depth that IFs can be nested, but an EVALUATE statement can specify any number of conditions.

Example 2. The following example is equivalent to the example above if there is a period after END-EVALUATE. See Example 3 below. It uses IF-THEN-ELSE statements instead of an EVALUATE statement.

```
IF HOURS-WORKED = 0
   PERFORM NO-PAY
ELSE IF HOURS-WORKED >= 1 AND <= 40
   PERFORM REG-PAY
   ELSE IF EXEMPT ='N'
          IF HOURS-WORKED >= 41 AND <= 80
            PERFORM OVERTIME-PAY
          ELSE PERFORM PAY-ERROR
        ELSE IF EXEMPT = 'Y'
               IF HOURS-WORKED >= 41 AND <= 80
                    PERFORM REG-PAY
               ELSE PERFORM PAY-ERROR
             ELSE PERFORM PAY-ERROR.
```

Example 3. The following example is also equivalent to the example above, but it uses the structured form of the IF-THEN-ELSE statement with the END-IF scope terminator:

```
IF HOURS-WORKED = 0
   PERFORM NO-PAY
ELSE
   IF HOURS-WORKED >= 1 AND <= 40
      PERFORM REG-PAY
   ELSE
      IF EXEMPT ='N'
         IF HOURS-WORKED >= 41 AND <= 80
            PERFORM OVERTIME-PAY
         ELSE
            PERFORM PAY-ERROR
         END-IF
      ELSE
         IF EXEMPT = 'Y'
            IF HOURS-WORKED >= 41 AND <= 80
               PERFORM REG-PAY
            ELSE
               PERFORM PAY-ERROR
            END-IF
         ELSE
            PERFORM PAY-ERROR
         END-IF
      END-IF
   END-IF
END-IF
```

HP COBOL II/XL evaluates the clauses in an EVALUATE statement in order. For fastest execution, order the clauses from most frequent value to least frequent value.

Explicit Scope Terminators

An explicit scope terminator is a keyword, END-*verb*, that terminates the scope of the last instance of the keyword *verb*. The explicit scope terminators are listed in the following table:

Table 3-1. The Scope Terminators

END-ACCEPT	END-IF	END-START
END-ADD	END-MULTIPLY	END-STRING
END-CALL	END-PERFORM	END-SUBTRACT
END-COMPUTE	END-READ	END-UNSTRING
END-DELETE	END-RETURN	END-WRITE
END-DIVIDE	END-REWRITE	
END-EVALUATE	END-SEARCH	

Explicit scope terminators help to eliminate logic errors caused by misplaced periods. With explicit scope terminators, periods are required only to terminate paragraphs in the PROCEDURE DIVISION.

Example 1. The following example shows the END-IF scope terminator. The first END-IF terminates the scope of IF PROCESS-2-OK. The second END-IF terminates the scope of IF PROCESS-1-OK.

```
IF PROCESS-1-OK THEN
  IF PROCESS-2-OK THEN
    MOVE 2 TO PROCESS-DATA-FLAG
  ELSE
    MOVE 1 TO PROCESS-DATA-FLAG
  END-IF
  PERFORM PROCESS-DATA
ELSE
  PERFORM PROCESS-1-ERROR-CHECK
END-IF
```

A conditional statement used with an explicit scope terminator is called a *delimited scope statement*. Unlike an ordinary conditional statement, a delimited scope statement is legal wherever an imperative statement is legal.

Example 2.

```
READ FILE-IN AT END
   ADD A TO B ON SIZE ERROR
      PERFORM OVERFLOW-ROUTINE
   END-ADD
   MOVE SPACES TO REC-IN.
```

The ADD statement with the ON SIZE ERROR phrase would be a conditional statement if not for the END-ADD, which terminates its scope and makes it a delimited scope statement. The ADD statement and the imperative statement MOVE make up the statement group following the conditional phrase AT END. Ordinary conditional statements are illegal in the statement group following a conditional phrase.

NOT Phrases

A NOT phrase specifies a set of statements to be executed if an exception condition does not occur. The NOT phrases are listed below:

```
NOT AT END
NOT AT END-OF-PAGE
NOT INVALID KEY
NOT ON EXCEPTION
NOT ON INPUT ERROR
NOT ON OVERFLOW
NOT ON SIZE ERROR
```

Using NOT phrases can make code more readable and sometimes more efficient.

Example. The following are functionally equivalent:

```
READ IN-FILE                          READ IN-FILE
   AT END MOVE 'YES' TO EOF.             AT END MOVE 'YES' TO EOF
IF EOF <> 'YES' THEN                     NOT AT END ADD 1 TO IN-CNT.
   ADD 1 TO IN-CNT.
```

The statements on the left perform two tests for every record read. The statement on the right performs one test for every record read. In this case, the NOT phrase makes the code more efficient as well as more readable.

NOT phrases used with I-O verbs execute only after a successful condition occurs. In the preceding example on the right, "`ADD 1 TO IN-CNT`" is not executed if a logic error exists.

PERFORM Statement Enhancements

The enhanced PERFORM statement can contain a list of statements rather than only procedure names if it ends with an END-PERFORM. This form of the PERFORM statement is called an *in-line* PERFORM statement.

Example 1. The following is an in-line PERFORM statement:

```
PERFORM 10 TIMES
   ADD A TO B
   ADD 1 TO A
END-PERFORM
```

Example 2. The in-line PERFORM statement can significantly reduce code fragmentation. It eliminates the need for short paragraphs whose only functions are to perform other paragraphs. The following two code fragments are equivalent:

```
PERFORM A100-CALC-A A-CNT TIMES.
          ⋮
A100-CALC-A.
   ADD C TO A (SUBA).
   ADD 1 TO SUBA.
        ⋮
```

```
PERFORM  A-CNT TIMES
   ADD C TO A (SUBA)
   ADD 1 TO SUBA
END-PERFORM.
```

The PERFORM on the left executes a separate paragraph in a different location in the program. The inline PERFORM on the right is functionally equivalent. By embedding the statements within the PERFORM, the program is much easier to read.

The enhanced PERFORM statement can also specify whether the UNTIL condition is to be tested before or after the statements or paragraphs have been executed.

Example 3. The following PERFORM statement tests EOF-FLAG and then performs READ-LOOP if EOF-FLAG is false:

```
PERFORM READ-LOOP
   WITH TEST BEFORE     BEFORE is the default.
   UNTIL EOF-FLAG
```

The following PERFORM statement performs READ-LOOP and then tests EOF-FLAG. If EOF-FLAG is false, it performs READ-LOOP again:

```
PERFORM READ-LOOP
   WITH TEST AFTER
   UNTIL EOF-FLAG
```

Example 4. In-line PERFORM statements can be nested. Nested in-line PERFORM statements can make your program more readable and less fragmented.

The following shows an example without nested in-line PERFORM statements:

```
      PERFORM PROC-NAME-1
        VARYING DEPARTMENT FROM FIRST-DEPT BY 1 UNTIL LAST-DEPARTMENT
        AFTER HOURS-PER-EMPLOYEE FROM FIRST-EMP BY 1 UNTIL LAST-EMPLOYEE.
      STOP RUN.
          ⋮
PROC-NAME-1.
          ⋮
```

The following shows an example with nested in-line PERFORM statements:

```
PERFORM VARYING DEPARTMENT FROM FIRST-DEPT BY 1 UNTIL LAST-DEPARTMENT
   {statement-group-1}
   PERFORM VARYING HOURS-PER-EMPLOYEE
   FROM FIRST-EMP BY 1 UNTIL LAST-EMPLOYEE
      {statement-group-2}
   END-PERFORM
   {statement-group-3}
END-PERFORM
```

With only *statement-group-2*, the second example is functionally equivalent to the first. With *statement-group-1* and *statement-group-3*, the second example is more powerful than the first.

USE GLOBAL AFTER ERROR PROCEDURE ON Statement

The USE GLOBAL AFTER ERROR PROCEDURE ON statement makes the scope of a USE procedure match the scope of the program that declares the USE procedure. That is, the statement applies to the program that contains it and to all programs directly or indirectly contained within that program.

A USE GLOBAL AFTER ERROR PROCEDURE ON statement specifies either a file open mode or the name of a GLOBAL file.

Using a File Open Mode. An example of the first case is the following statement:

```
USE GLOBAL AFTER ERROR PROCEDURE ON INPUT.
```

If the program containing this statement, or any program contained in that program, encounters an error while reading any file that is open for input, the USE procedure is invoked. Thus, file error handling is standardized in the outermost program, even if the errors occur on files that are local to inner programs and invisible to the outermost program. (In the example in the next section, "When to Use Nested Programs and GLOBAL Data," only the inner programs invoke the GLOBAL USE AFTER ERROR PROCEDURE ON INPUT statement in the outer program, because only they have files that are open for input.)

Using a GLOBAL File. An example of the second case is the following statement, where FILE-A is a GLOBAL file:

```
USE GLOBAL AFTER ERROR PROCEDURE ON FILE-A
```

If a program containing this statement, or any program contained in that program, encounters an error while accessing FILE-A in any way, the USE procedure is invoked. Thus, the error handling for the GLOBAL file FILE-A need only be coded once, even though errors on FILE-A may be encountered in other (contained) programs.

If more than one USE procedure applies to a situation, the first one found is executed. The search for this "first found" USE procedure begins in the program where the situation arises and proceeds outward through the enclosing programs.

Note An EXIT PROGRAM statement executed directly or indirectly within a GLOBAL USE procedure has undefined results.

When to Use Nested Programs and GLOBAL Data

The structured programming features described in the previous section, "Structured Programming," are part of ANSI85 COBOL. They allow you to divide your application into programs whose nesting hierarchy and data organization express and document your programming approach. This clarifies the logical structure of your program, making it easier to understand, debug, and maintain.

HP COBOL II/XL has always had divisions, paragraphs, sections, and the PERFORM statement to organize code. However, the relationship between data and code in complex COBOL programs has been obscured by the size of the DATA DIVISION and the complexity of the code. Nested programs offer a solution to this problem, a way to associate data with the code that uses it. Data that is used by many programs in an application can be declared GLOBAL in the DATA DIVISION of the outermost program. Data that is used by one program can be declared in that program's DATA DIVISION only.

Nested programs are also appropriate for applications where complex tasks are composed of smaller tasks. The complex task can be initiated by the outer program and the smaller tasks can be performed by programs nested within it. And the smaller tasks can be broken down into relatively simple paragraphs and sections. The nesting structure helps document the data and logic dependencies of the program as a whole.

A good way to decide whether to break an application down into nested programs (instead of just paragraphs or sections) is to first decide on the clearest way to organize the data. If a task is fairly complex and requires a set of data items and files that other sections of the program will not need, then the task is a candidate for nested programming.

Example - A Payroll Application

The following simplified payroll program illustrates the approach discussed above, using nested programs, GLOBAL data items and files, and GLOBAL USE procedures.

The nested programs break the payroll application into logical units with distinct functionality. Note that one of the nested programs is declared COMMON. In this simple example, it does not need to be COMMON, but in a realistic payroll implementation, this would be appropriate (see the comments in the program below).

The payroll program contains both local and GLOBAL files. Local files are used when only one program needs to access their data; GLOBAL files, when all programs do.

A GLOBAL USE procedure is declared ON INPUT. Only local files are opened for input, so this USE procedure is only invoked by local files. Because this USE procedure is GLOBAL, this code only needs to appear in one place.

The comments in the program itself explain the logic and nesting structure of the program in more detail.

```
**************************
*PROGRAM-ID. PAYROLL.
* Payroll declares and opens two global files CURR-PAY-REC and
* EMPLOYEE-INFO.  For each record in CURR-PAY-REC it calls
* GET-CURR-GROSS and GET-CURR-DEDUCTIONS. It also updates the
* year-to-date payroll fields in EMPLOYEE-INFO.
* It declares a GLOBAL use procedure for INPUT mode, which will
* be used when local files with tax and pay scale tables
* (opened only in input mode) encounter errors.
*
*     ****************************
*     *  PROGRAM-ID. GET-CURR-GROSS.
*     *  {Calculates current gross, using a local file PAY-FILE
*     *   which contains the pay rates.}
*     *  END PROGRAM GET-CURR-GROSS.
*     ****************************
*
*     ****************************
*     *  PROGRAM-ID. GET-CURR-DEDUCTIONS.
*     *  {Calculates current deductions, using a local file TAX-FILE.
*     *   For simplicity only social security tax is calculated. The
*     *   complex calculations of a realistic payroll program could
*     *   be carried out with a set of local data declarations.}
*     *  END PROGRAM GET-CURR-DEDUCTIONS.
*     ****************************
*
*     ****************************
*     *  PROGRAM-ID. HASHED-READ-ON-EMPLOYEE-FILE IS COMMON.
*     *   Passed a social security number, this routine makes a
*     *   relative key and executes either a read or rewrite.
*     *   This program is COMMON. It is callable by any program
*     *   nested within PAYROLL. (Only the outermost program calls
*     *   it here but a realistic payroll program might call it
*     *   to update yearly vacation, sick leave, tax status etc.)
*     *  END PROGRAM HASHED-READ-ON-EMPLOYEE-FILE.
*     ****************************
*
* END PROGRAM PAYROLL.
****************************
```

```
$PAGE "PAYROLL"
    IDENTIFICATION DIVISION.
    PROGRAM-ID. PAYROLL.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    SELECT CURR-PAY-FILE ASSIGN TO "CURRPAY"
           ORGANIZATION IS SEQUENTIAL.
    SELECT EMPLOYEE-INFO ASSIGN TO "EMPINFO"
           ORGANIZATION IS RELATIVE
           ACCESS IS RANDOM
           RELATIVE KEY IS EMP-INFO-KEY
           FILE STATUS IS FILE-STATUS.
    DATA DIVISION.
    FILE SECTION.
*   When the FD is GLOBAL, all subordinate records are also GLOBAL.
    FD  CURR-PAY-FILE IS GLOBAL.
    01  CURR-PAY-REC.
       05  CURR-NAME               PICTURE X(20).
       05  CURR-SS-NO              PICTURE X(9).
       05  CURR-HOURS              PICTURE 999.
       05  CURR-GROSS              PICTURE $$$,$$$.99.
       05  CURR-DEDUCTIONS         PICTURE $$$,$$$.99.
    FD EMPLOYEE-INFO IS GLOBAL.
    01  EMPLOYEE-REC.
       05  EMP-SS-NO                   PICTURE X(9).
       05  EMP-JOB-DESCRIPTOR          PICTURE 99.
       05  EMP-YEARLY-GROSS            PICTURE 9(6)V99
                                       USAGE PACKED-DECIMAL.
       05  EMP-YEARLY-DEDUCTIONS       PICTURE 9(6)V99
                                       USAGE PACKED-DECIMAL.
    WORKING-STORAGE SECTION.
    01 EMP-INFO IS GLOBAL.
       05  UNIQUE-KEY                  PIC 999.
           88   NO-UNIQUE-KEY          VALUE 999.
    01 EMP-INFO-KEY REDEFINES EMP-INFO IS GLOBAL PIC 9(3).
    01 FILE-STATUS                     IS GLOBAL PIC XX.
    01 FILE-NAME          IS GLOBAL PIC X(10) VALUE SPACES.
```

```
       PROCEDURE DIVISION.
       DECLARATIVES.
       GLOBAL-USE-PROC SECTION.
          USE GLOBAL AFTER STANDARD ERROR PROCEDURE ON INPUT.
       GLOBAL-USE.
      *   This will be executed when local files TAX-RATES and
      *   PAY-RATES encounter an error.
          DISPLAY FILE-NAME, " Status is ", FILE-STATUS.
       END DECLARATIVES.

       PROCESS-PAYROLL SECTION.
       OPEN-FILES.
          OPEN I-O CURR-PAY-FILE
          OPEN I-O EMPLOYEE-INFO.
       READ-PAY-FILE.
          READ CURR-PAY-FILE
          AT END
             CLOSE CURR-PAY-FILE
          NOT AT END
             CALL "HASHED-READ-ON-EMPLOYEE-FILE" USING CURR-SS-NO
             CALL "GET-CURR-GROSS"
             CALL "GET-CURR-DEDUCTIONS"
             DISPLAY CURR-NAME," ", CURR-SS-NO," ", CURR-HOURS,
                     " ",CURR-GROSS," ", CURR-DEDUCTIONS
             WRITE CURR-PAY-REC
             CALL "HASHED-REWRITE-ON-EMPLOYEE-FILE" USING CURR-SS-NO
             DISPLAY EMP-SS-NO," ",EMP-JOB-DESCRIPTOR," ",
                     EMP-YEARLY-GROSS," ",EMP-YEARLY-DEDUCTIONS
             GO READ-PAY-FILE
          END-READ
       CLOSE EMPLOYEE-INFO
       STOP RUN.
```

```
$PAGE "GET-CURR-GROSS"
$CONTROL DYNAMIC
    IDENTIFICATION DIVISION.
    PROGRAM-ID. GET-CURR-GROSS.
*   USE EMP-JOB-DESCRIPTOR TO INDEX TABLE-OF-PAY TO GET WAGE RATE
*   AND CALCULATE CURRENT GROSS SALARY.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    SELECT PAY-RATES ASSIGN TO "PAYRATES"
    FILE STATUS IS FILE-STATUS.
    DATA DIVISION.
    FILE SECTION.
*   PAY-RATES is LOCAL to this program.
    FD PAY-RATES.
    01  TABLE-OF-PAY.
      05  RATES          OCCURS 99 TIMES.
        10  HOURLY-PAY   PICTURE 9999V99 USAGE PACKED-DECIMAL.
        10  REDEF-HOURLY REDEFINES HOURLY-PAY.
            15  SALARY   PICTURE 9999V99 USAGE PACKED-DECIMAL.
    WORKING-STORAGE SECTION.
    01  RATE             PICTURE 9999V99 USAGE PACKED-DECIMAL.
    01  GROSS            PICTURE 9(6)V99 USAGE PACKED-DECIMAL.
    01  JOB-CLASS                        PICTURE 99.
        88  NON-EXEMPT                   VALUE 0 THRU 50.
        88  EXEMPT                       VALUE 51 THRU 99.
    01  OVERTIME                         PICTURE 999.
        88  WORKED-OVERTIME              VALUE 41 THRU 100.
    PROCEDURE DIVISION.
    OPEN-LOCAL-FILE.
       MOVE "PAYRATES" TO FILE-NAME.
       OPEN INPUT PAY-RATES
       READ PAY-RATES.
    UPDATE-GROSS-PAY.
       MOVE EMP-JOB-DESCRIPTOR TO JOB-CLASS
       EVALUATE EXEMPT
          WHEN TRUE   PERFORM SALARIED-LABOR
          WHEN FALSE  PERFORM HOURLY-LABOR
       END-EVALUATE
       COMPUTE EMP-YEARLY-GROSS = EMP-YEARLY-GROSS + GROSS
       MOVE GROSS TO CURR-GROSS
    CLOSE-LOCAL-FILE.
       CLOSE PAY-RATES.
       EXIT PROGRAM.
```

```
       HOURLY-LABOR.
          MOVE HOURLY-PAY (EMP-JOB-DESCRIPTOR) TO RATE
          MOVE CURR-HOURS TO OVERTIME
          EVALUATE WORKED-OVERTIME
             WHEN FALSE
                COMPUTE GROSS = CURR-HOURS * RATE
             WHEN TRUE
                COMPUTE GROSS = 40 * RATE +
                   (CURR-HOURS - 40 * RATE * 1.5)
          END-EVALUATE.

       SALARIED-LABOR.
          MOVE SALARY (EMP-JOB-DESCRIPTOR) TO GROSS.

       END PROGRAM GET-CURR-GROSS.
```

```
$PAGE "GET-CURR-DEDUCTIONS"
    IDENTIFICATION DIVISION.
    PROGRAM-ID. GET-CURR-DEDUCTIONS.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    SELECT TAX-RATES ASSIGN TO "TAXRATES"
    FILE STATUS IS FILE-STATUS.
    DATA DIVISION.
    FILE SECTION.
*   TAX-RATES is LOCAL to this program.
*   For simplicity, only social security tax is calculated.
    FD TAX-RATES.
    01  TABLE-OF-TAXES.
        05  FICA-TAX-RATE PICTURE 9999V999 USAGE PACKED-DECIMAL.
    WORKING-STORAGE SECTION.
    01  GROSS            PICTURE 9(6)V99 USAGE PACKED-DECIMAL.
    01  DEDUCTIONS       PICTURE 9(6)V99 USAGE PACKED-DECIMAL.
    PROCEDURE DIVISION.
    OPEN-LOCAL-FILE.
       MOVE "TAXRATES" TO FILE-NAME
       OPEN INPUT TAX-RATES
       READ TAX-RATES.
    UPDATE-DEDUCTIONS.
       MOVE CURR-GROSS TO GROSS
       COMPUTE DEDUCTIONS = GROSS * FICA-TAX-RATE
       COMPUTE EMP-YEARLY-DEDUCTIONS =
          EMP-YEARLY-DEDUCTIONS + DEDUCTIONS
       MOVE DEDUCTIONS TO CURR-DEDUCTIONS.
    CLOSE-LOCAL-FILE.
       CLOSE TAX-RATES.
    END PROGRAM GET-CURR-DEDUCTIONS.
```

```
$PAGE "HASHED-ACCESS-ON-EMPLOYEE-FILE"
    IDENTIFICATION DIVISION.
    PROGRAM-ID. HASHED-READ-ON-EMPLOYEE-FILE IS COMMON.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 KEY-SWITCH                      PIC X.
       88  SUCCESS                    VALUE "Y".
       88  RESET-SWITCH               VALUE "N".
    LINKAGE SECTION.
    01 SS-NO                           PICTURE X(9).

    PROCEDURE DIVISION USING SS-NO.

    BEGIN-HASHED-READ.
       SET RESET-SWITCH TO TRUE
       MOVE SS-NO(1:3) TO EMP-INFO-KEY
       PERFORM UNTIL SUCCESS
          READ EMPLOYEE-INFO
             INVALID KEY         ADD 1 TO UNIQUE-KEY
             NOT INVALID KEY     IF EMP-SS-NO = SS-NO THEN
                                     SET SUCCESS TO TRUE
                                 END-IF
          END-READ
       END-PERFORM
       EXIT PROGRAM.

    ENTRY "HASHED-REWRITE-ON-EMPLOYEE-FILE" USING SS-NO.
    BEGIN-HASHED-REWRITE.
       IF EMP-SS-NO  SS-NO THEN
          DISPLAY "HASHING SCHEME REQUIRES READ BEFORE REWRITE"
       ELSE
          REWRITE EMPLOYEE-REC
          INVALID KEY
             DISPLAY "Employee-file Status is ", FILE-STATUS
          NOT INVALID KEY
             EXIT PROGRAM
       END-IF.
    END PROGRAM HASHED-READ-ON-EMPLOYEE-FILE.

    END PROGRAM PAYROLL.
```

Run-Time Efficiency

You can improve your program's run-time efficiency with the following:

- An improved algorithm.

  This is the most important way you can improve your program's efficiency. Neither control options nor the optimizer can make up for a slow algorithm. A program that uses a binary search (without control options or optimization) is still faster than a program that uses a linear search (with control options and optimization).

- Coding heuristics.
- Control options.
- The optimizer.

This section discusses the last three ways of improving rum-time efficiency.

Note Coding heuristics and the optimizer do not significantly improve the performance of I-O-bound programs.

Coding Heuristics

The following coding heuristics for run-time efficiency are guidelines, not rules, for writing programs that run faster. They do not always work, but programmers have learned from experience that they usually do.

- Put variables that are referenced most often (such as array subscripts and counters) at the beginning of the WORKING-STORAGE SECTION in the main program and nondynamic subprograms. Put them at the end of the WORKING-STORAGE SECTION in each dynamic subprogram.
- Avoid conversion of data to different types. If two or more variables are operands in the same operation, declare them to be of the same type. It is more efficient to have operands of the same type than to make one of several operands "faster." See the examples below.
- If fields are often used together as operands in arithmetic statements, it is more efficient to define them with the same PICTURE clause.
- The following examples illustrate the above two points.
  - The first of the following COMPUTE statements is faster than the other two because no conversion is necessary and the intermediate result is the same as the receiving operand, DISPLAY-4. The second COMPUTE requires a conversion from BINARY to DISPLAY because the receiving item is DISPLAY. The third statement requires a conversion from BINARY to PACKED-DECIMAL because the receiving item is PACKED-DECIMAL. These conversions take many machine instructions.

```
01  DISPLAY-4       PIC S9(4).                  4 bytes long.
01  BINARY-4        PIC S9(4) BINARY.           2 bytes long.
01  DECIMAL-9       PIC S9(9) PACKED-DECIMAL.   5 bytes long.
     ⋮
COMPUTE DISPLAY-4 = DISPLAY-4 + DISPLAY-4.
COMPUTE DISPLAY-4 = BINARY-4 + BINARY-4.
COMPUTE DECIMAL-9 = BINARY-4 + BINARY-4.
```

- Calculations involving multiplication, division, and exponentiation can require conversions for intermediate results. When the intermediate results of BINARY operands exceed 18 digits, the operands are converted to PACKED-DECIMAL. This takes many extra instructions.

  The first of the following COMPUTE statements is faster then the second because the intermediate result is 18 digits. The second requires conversion to PACKED-DECIMAL because the intermediate result is 20 digits. The result of the multiplication is then converted back to BINARY.

```
01  BINARY-9       PIC S9(9) BINARY.       4 bytes long.
01  BINARY-10      PIC S9(10) BINARY.      8 bytes long.
      ⋮
COMPUTE BINARY-9 = BINARY-9 * BINARY-9.
COMPUTE BINARY-9 = BINARY-10 * BINARY-10.
```

- The first of the following COMPUTE statements is faster because the intermediate result is 16 bits. The intermediate result of the second COMPUTE is 32 bits so the operands must be converted to 32-bit values.

```
01  BINARY-2        PIC S9(2) BINARY.     16 bits long.
01  BINARY-3        PIC S9(3) BINARY.     16 bits long.
01  BINARY-4        PIC S9(4) BINARY.     16 bits long.
      ⋮
COMPUTE BINARY-4 = BINARY-2 * BINARY-2.
COMPUTE BINARY-4 = BINARY-2 * BINARY-3.
```

- When you MOVE BINARY data items from a larger field to a smaller field, many instructions are required to truncate the data. The first of the following MOVE statements is faster than the second because the data must be truncated in the second MOVE but not the first.

```
01  BINARY-3        PIC S9(3) BINARY.     16 bits long.
01  BINARY-4        PIC S9(4) BINARY.     16 bits long.
      ⋮
MOVE BINARY-4 TO BINARY-4.
MOVE BINARY-4 TO BINARY-3.
```

- If a variable is a subscript or a varying identifier in a PERFORM loop, declare it to be of the type PIC S9(9) BINARY SYNC.
- When coding the UNTIL condition in a loop, keep this in mind: the comparisons *equal* and *not equal* are faster than the comparisons *less than* and *greater than*.
- Compile subprograms with the SUBPROGRAM or ANSISUB control option. Calls to such subprograms execute faster than calls to subprograms compiled with PROGRAM-ID *identifier* IS INITIAL or the DYNAMIC control option, which require that the subprogram data be reinitialized whenever the program is called.

  For less code, use the control option SUBPROGRAM instead of ANSISUB. Performance is the same, but an ANSISUB subprogram contains extra code that reinitializes its data if the main program executes a CANCEL statement.
- If paragraphs are performed close together timewise, put them close together physically in your source code.
- If a paragraph is performed from only one place, use an in-line PERFORM statement for it. This applies especially to loops.
- Use NOT phrases to minimize checking. NOT AT END is especially useful. See the example in "NOT Phrases."
- Do not pass parameters BY CONTENT.
- Do not specify the ON EXCEPTION or ON OVERFLOW phrase in a CALL statement when you use a literal to specify the program name. If you do, the program is not bound to the subprogram until run time. This slows the program by approximately .01 second per CALL, and the loader cannot detect missing subprograms.
- Do not use NEXT SENTENCE as the ELSE clause in an IF statement. Use CONTINUE or END-IF, or reverse the sense of the condition.
- Computation is fastest with the following types of operands, listed from the fastest to the slowest:
  - PIC S9(9) BINARY, SYNCHRONIZED.
  - PIC S9(4) BINARY, SYNCHRONIZED.
  - PACKED-DECIMAL, the fewer digits the faster the computation.
  - Numeric DISPLAY, the fewer digits the faster the computation.

Computation is faster with signed numbers than with unsigned numbers.

Coding Heuristics when Calling COBOL Functions

The following are guidelines when your program calls COBOL functions. For more information on the COBOL functions, see Chapter 10, "COBOL Functions," in the *HP COBOL II/XL Reference Manual*.

- Some of the functions are implemented as calls to run-time libraries. The rest are implemented simply as inline code. Inline functions are generally faster than functions in the run-time library. You might find that coding your own routine for some library functions is faster than calling the COBOL function.
- The following functions convert the parameter values to floating point values to calculate the function result. These functions will execute faster on systems that have a floating point coprocessor. Use the ROUNDED phrase when precision of these function values is important.
  - ACOS, ASIN, ATAN, COS, SIN, TAN.
  - LOG, LOG10, RANDOM, SQRT, SUM.
  - MAX, MIN (on numeric operands).
  - NUMVAL, NUMVAL-C.
  - ORD-MAX, ORD-MIN.
  - ANNUITY, MEAN, MEDIAN, MIDRANGE, PRESENT-VALUE, RANGE, STANDARD-DEVIATION, VARIANCE.
- The precision of functions that convert the parameter values to floating point values is limited to 15 significant digits. Also, fractional values may have rounding errors even if the total size of the argument is less than or equal to 15 digits. Use of equality comparisons, as shown below, are not recommended.

  Not recommended:

```
IF FUNCTION COS(ANGLE-RADIANS) = 0.1 THEN
   PERFORM P1
END-IF
```

  Recommended alternative:

```
COMPUTE ROUNDED COS-NUM = FUNCTION COS(ANGLE-RADIANS)
IF COS-NUM > 0.1 THEN PERFORM P1.
```

  where COS-NUM is defined as follows:

```
01 COS-NUM   PIC S9V9 COMP.
```

  Another alternative:

```
IF FUNCTION COS(ANGLE-RADIANS) >= .0999 AND <= 0.1001 THEN
   PERFORM P1
END-IF
```

Control Options

These control options make a program run faster:

- OPTIMIZE=1, which invokes the optimizer. See the section, "The Optimizer", for details.
- OPTFEATURES=LINKALIGNED[16], which generates code that accesses variables in the LINKAGE SECTION more efficiently. (If the called program specifies OPTFEATURES=LINKALIGNED[16], have the calling program specify OPTFEATURES=CALLALIGNED[16].)
- SYNC32, which allows the compiler to align variables along the optimum boundaries.

These control options make a program run more slowly:

- VALIDATE, which takes extra time to check that the digits and signs of numeric items are valid.
- BOUNDS, which generates and executes extra code to check ranges.
- SYNC16, which specifies an alignment that is not optimum for Series 900 systems.
- ANSISORT, which prevents SORT from reading or writing files directly.
- SYMDEBUG, which generates Symbolic Debug information to be executed in the program file.

  (If a source program was compiled with SYMDEBUG, you can link the object module with the NODEBUG option, causing the program to ignore the Symbolic Debug information and improving its execution speed).

This control option makes a program compile more slowly:

- CALLINTRINSIC, which causes the compiler to check the file SYSINTR for each call literal.

The Optimizer

The optimizer is an optional part of the compiler that modifies your code so that it uses machine resources more efficiently, using less space and running faster. Note that the optimizer improves your code, not your algorithm. Optimization is no substitute for improving your algorithm. A good, unoptimized algorithm is always better than a poor, optimized one.

You can compile your program with level one optimization by compiling it with the control option OPTIMIZE=1.

The advantages of level one optimization are:

- The program is approximately 3.1% smaller.
- The program runs 2.8% to 4.4% faster. (Programs that use the Central Processor Unit intensively and those that do relatively little I-O benefit most.)

The disadvantages of level one optimization are:

- The program compiles approximately 10% more slowly.
- The symbolic debugger cannot be used with program.
- The statement numbers are not visible when using DEBUG or from trap messages (see the example in Chapter 7 for details).

Level two optimization is not available for HP COBOL II/XL programs for the following reasons:

- The most common COBOL data type is the multibyte string, which is difficult to optimize. (The easiest types to optimize are level 01 or 77, 16- or 32-bit, binary data types.)
- HP COBOL II/XL programs call millicode routines far more often than non-COBOL programs do, and the optimizer does not optimize across routine calls. Level two optimization for COBOL would not have improved performance enough to be worth the effort, which was spent on improving millicode routines and code generation instead.

Millicode Routines

Millicode routines are assembly language routines that deliver high performance for common COBOL operations such as move and compare. It supports COBOL operations on MPE XL the way microcode supports them on MPE V.

Millicode routines are very specialized, tuned to provide optimal performance for specific data types of specific lengths. Based on operation and data types and lengths, the COBOL compiler calls the appropriate millicode routines.

The calling convention for a millicode routine differs from that of an ordinary routine in the following ways:

- A millicode call requires fewer registers to be saved across a call, making it faster than an ordinary call.
- A millicode call uses general register 31 as the return register, rather than general register 2.

When to Use the Optimizer

Compile your program with optimization only after you have debugged it. The optimizer can transform legal programs only.

Once you have compiled your program with optimization, you cannot use the symbolic debugger to debug it. This is because debug information will be missing from it. The compiler does not generate debug information and perform optimizations at the same time.

You can still use DEBUG on your program after you have compiled it with optimization; however, the statement numbers will not appear in the code.

Transformations

The five level one optimizer transformations are:

1. **Basic block optimization.**

   The optimizer reads the machine code (specifically, the machine instruction list). When it reads a branch instruction, it creates a *basic block* of code. It joins two basic blocks into one *extended block* in the following two cases:

   a. If it can remove the branch instruction and append the "branched to" block to the "branched from" block.

   b. If the basic blocks are logically related and can be optimized as a single unit.

   Basic block optimization has these three components:

   a. Removal of common expressions in basic blocks.

      If more than one expression assigns the same value to the same field within the block, the optimizer removes all but the first expression.

   b. Removal and optimization of load and copy instructions in basic blocks.

   c. Optimization of elementary branches in basic and extended blocks. (Removal of unnecessary branches and more efficient arrangement of necessary branches.)

2. **Instruction scheduling.**

   Instruction scheduling reorders the instructions within a basic block to accomplish the following:

   a. Minimize load and store instructions.

   b. Optimize the scheduling of instructions that follow branches.

3. **Dead code elimination.**

   Dead code elimination is the removal of all code that will never be executed.

4. **Branch optimization.**

   Branch optimization traces the flow of IF-THEN-ELSE-IF ... statements and reorganizes them more efficiently.

5. **Peephole optimization.**

   Peephole optimization substitutes simpler, equivalent instruction patterns for more complex ones.

Portability

Portability applies to both programs and files. The more portable your program or file is, the less you need to modify it to use it on machines other than the one for which you originally created it.

This chapter assumes that you are creating your program or file for one HP 3000 machine and then transporting it to another HP 3000 machine or a non-HP machine.

To make your program more portable in both cases, do the following:

- Make your program correct.

  An incorrect program that works on one machine may not work (or work the same) on another machine. It may not even work on the same machine if you recompile it with a later version of the COBOL compiler.

  To this end, do not use the following in your program:

  - Uninitalized variables.
  - Illegal PERFORM statements (such as recursive PERFORM statements or two or more PERFORM statements with a common exit point).
  - Branches out of the range of PERFORM statements.

- Do not depend on specific error recovery behavior that the *HP COBOL II/XL Reference Manual* does not specify. For example:

  - Do not depend on the truncation of digits when there is no ON SIZE ERROR phrase.
  - Do not depend on a specific result when the value of a data item does not match its PICTURE string (for example, unsigned data in a signed field).

The rest of this section covers these portability issues:

- Portability between HP 3000 Architectures.
- Portability between HP 3000 and non-HP machines.
- Cross-development.
- HP extensions.

Portability Between HP 3000 Architectures

If you are writing your program for one HP 3000 machine but also intend to run it on another HP 3000 machine, you should do the following:

- Use the CALL INTRINSIC form of the verb to call an intrinsic. This tells the compiler an intrinsic is being called and allows it to adapt the call to a specific operating system.
- Do not use the pseudo-intrinsics .LOC. or .LEN. which call operating system intrinsics. These are HP Extensions and are highly machine dependent. You can use the LENGTH COBOL function in place of .LEN..
- Do not name your program the name of an intrinsic.
- Do not call COBOLLOCK or COBOLUNLOCK which are obsolete COBOL 68 features. Instead, either use the statements EXCLUSIVE and UN-EXCLUSIVE or call the intrinsics FLOCK and FUNLOCK with CALL INTRINSIC.
- Do not declare the identifier of a CALL identifier statement to be numeric.
- Do not assume that external names will be truncated or dehyphenated. (External names are compiler-generated names that are recognized outside the program. Chapter 4 explains them.)
- Do not write an illegal program, even if it works on the first machine for which you are writing the program. It may not work the same way on another machine. (Examples of illegal things that may work differently on different machines are: branching out of PERFORM paragraphs, PERFORM statements with common exit points, and indirectly recursive PERFORM statements.)
- Do not operate on illegal data, even if the results are satisfactory on the first machine for which you are writing the program. The results may cause problems on another machine. (Examples of illegal data that may cause problems on some machines but not others are: COMP field overflow, illegal data in PACKED-DECIMAL or numeric DISPLAY fields, and signed data in unsigned fields.)
- Do not assume that DISPLAY statement output will have exactly the same format on different machines.
- Initialize all data items before using them.
- Do not call unsupported or privileged mode intrinsics.
- If you are creating a file for one HP 3000 machine but also intend to use it on another HP 3000 machine, do not put indexed or SYNC data items in the file.

Portability Between HP 3000 and Non-HP Machines

If you are writing your program for an HP 3000 machine but also intend to run it on a non-HP machine, you should do the following:

- Use only ANSI Standard features.
- Do not call system intrinsics directly.

To ensure that your program conforms to the 1985 ANSI COBOL Standard, you can compile your program with the following $CONTROL options: ANSIPARM, ANSISORT, POST85, and STDWARN.

The COBOL standard specifies that the following features are implementation-defined. Before you use them in your program, check the specifications of the target machine to ensure that your program will run correctly on that machine.

- Computer name.
- Function names ASCII, EBCDIC, and EBCDIK in the SPECIAL-NAMES paragraph.
- Carriage control codes C01 through C16.
- NO SPACE CONTROL.
- TOP.
- Radix of data representation.
- Default representation and position of the sign for numeric data.
- Data alignment.
- External switches SW0 through SW15.
- Area B size.
- Default collating sequences.
- Correspondence between STANDARD-1 and the native character set.
- Default for a record (variable or fixed) when the program contains no RECORD clause or when the RECORD clause specifies a range of characters.
- Size and representation of index names and index data items.
- Specific positioning and generation of implicit FILLER in the SYNCHRONIZED clause.
- Whether data items that do not specify USAGE DISPLAY are automatically aligned.
- Precise effect of USAGE BINARY, USAGE PACKED-DECIMAL, USAGE COMPUTATIONAL, and USAGE INDEX on alignment and representation.
- Methods of evaluating arithmetic expressions.
- In the ACCEPT and DISPLAY statements: mnemonic name, size of data transfer, data conversion (if necessary), and standard device.
- Action after the USE PROCEDURE is executed for an error condition.
- Value of x in $9x$ file status codes.
- Format and meaning of *file-info* in the ASSIGN clause.
- Format of file labels.
- Rules for calling non-COBOL subprograms.

If you are creating a file for an HP 3000 machine but also intend to use it on a non-HP machine, you should do the following:

- Never put indexed data items in the file.
- Do not use SYNC in the file unless the compiler on the non-HP machine uses 16- or 32-bit synchronization for COMP and BINARY items. In that case, compile your program with the control option SYNC16 or SYNC32.
- Specify storage format with USAGE DISPLAY unless the non-HP machine stores COMP, BINARY, and PACKED-DECIMAL data the same way that the HP machine on which you are creating the file stores it.
- Realize that although SIGN IS SEPARATE is always portable, operations on such items are very slow.
- Use ASSIGN clauses.

Cross-Development

Cross-development is the development of a program on an MPE XL system for the MPE V system, or vice versa. In both cases, you should do the following:

- Consult the *HP COBOL II Migration Guide* and avoid features that are different on the two systems.
- Be sure that the COBCNTL.PUB.SYS files on both systems specify the same SYNC option, preferably SYNC32.

If you are developing a program on an MPE XL system and intend to run it on an MPE V system, use the Compatibility Mode compiler. It generates code that runs on the MPE V system. It also flags features that are not available on MPE V.

If you are developing a program on an MPE V system and intend to run it on an MPE XL system in Native Mode, be sure that the program and its data are portable, because you must recompile the program on MPE XL.

Note Ensure that all variables are properly initialized. Uninitialized variables that did not cause problems on MPE V systems may cause programs to abort on MPE XL systems.

HP Extensions

If your program uses HP extensions, you cannot transport it to non-HP computers. The following are HP extensions to ANSI COBOL 1985:

- NOLIST phrase in the COPY statement.
- USAGE COMP-3 (the standard equivalent is USAGE PACKED-DECIMAL).
- Intrinsic relation condition.
- Random access files.
- The USING phrase of the ASSIGN clause.
- WITH DUPLICATES phrase in the RECORD KEY clause of the SELECT statement.
- REMARKS paragraph in the IDENTIFICATION DIVISION.
- Abbreviation ID for "IDENTIFICATION".
- Ability to have a section name that is not followed by a paragraph name.
- Qualified index names.
- Ability to specify the language in the SORT statement.
- ACCEPT FREE form of the ACCEPT statement.
- SEEK statement.
- EXCLUSIVE statement.
- UN-EXCLUSIVE statement.
- CALL statement extensions. These are the INTRINSIC phrase, the GIVING clause, the symbol "@" or "\" before a parameter.
- The RETURN-CODE special register.
- ENTRY statement (secondary entry points).
- EXAMINE statement.
- GOBACK statement.
- Key name is not required to be on the left-hand side of the SEARCH ALL condition.
- Index data items in relational conditions.
- Special registers TALLY, CURRENT-DATE, TIME-OF-DAY, and WHEN-COMPILED.
- Nonnumeric literals longer than 160 characters. (The limit is 255 characters per nonnumeric literal)
- Octal literals.
- Interchangeability of single and double quotes.
- Inequality operator (<>).
- Use of the USE AFTER STANDARD BEGINNING statement to process user labels for files.
- Native Language Support using $CONTROL NLS.
- Nested COPY statements.

4

Subprograms and Intrinsics

Introduction

A *subprogram* is a routine that either is not in the program that calls it, or is nested within another program. It is the object of a CALL statement. Its source language can be the same as that of the calling program, or different.

An *intrinsic* is a system-supplied procedure, an external interface to the operating system or subsystem services that can be called through the intrinsic mechanism. The intrinsic mechanism checks the types and bounds of parameter values before using them. An intrinsic is not different from a subprogram that you write yourself, except that the details of its task are invisible to you.

This chapter explains the following:

- External names, which apply to subprograms, intrinsics, and data.
- Internal names, which apply to nested programs.
- Locality set names.
- Chunk names.
- Data alignment on MPE XL (relevant to parameter alignment).
- How COBOL checks actual parameters against their formal counterparts.
- How COBOL passes actual parameters to subprograms and intrinsics.
- When subprogram and intrinsic calls are bound to their definitions.
- How your COBOL program, compiled in Native Mode, can use switch stubs to call subprograms compiled in Compatibility Mode.
- How your COBOL program can call subprograms written in COBOL.
- How your COBOL program can call subprograms written in other languages.
- How your COBOL program can share EXTERNAL data items and files with other programs.
- Briefly, how your COBOL program can share GLOBAL data items and files with other programs (but see Chapter 3 for details).
- How your COBOL program can call intrinsics.

External Names

An *external name* is a compiler-generated name that is recognized outside the program. The compiler generates an external name for each of the following:

- *program-id*. †
- ENTRY statement literal. †
- EXTERNAL data item or file.
- Program name that a CALL or CANCEL statement specifies.

† An external name is only generated for a *program-id* or ENTRY statement literal of a separately compiled program. Programs nested within other programs are not separately compiled, and their names are never external.

The compiler forms the external name from the name in the program as follows:

- Converts each hyphen (-) to an underscore (_), unless the original name begins with a backslash (\).
- Changes uppercase letters to lowercase letters, unless the original name begins with a backslash (\).
- Truncates the original name to 30 characters if it is longer than 30 characters.

If you compile the program with the control option CMCALL (Compatibility Mode CALL), the compiler does the following unless the name begins with a backslash (\):

1. Strips hyphens from the name.
2. Changes uppercase letters to lowercase letters.
3. Truncates the name to 15 characters.

Use the CMCALL control option for the following:

- Programs that call subprograms written in languages that depend on 1 and 3 above.
- Native Mode programs that call Compatibility Mode subprograms.

The above rules do not apply to intrinsics, whose external names are supplied through the intrinsic mechanism in the file SYSINTR.PUB.SYS.

Example

The following are example program names and their corresponding external names:

Internal Name	External Name
`Sub-Total`	`sub_total`
`\Sub-Total`	`Sub-Total`
`Name-Longer-Than-Thirty-Characters`	`name_longer_than_thirty_charac`

Internal Names

An *internal name* is a compiler-generated name that is not recognized outside the program. The compiler generates an internal name for each:

- *program-id* of a nested program.
- ENTRY statement literal of a nested program.
- Nested program name that a CALL or CANCEL statement specifies.
- Nested procedure for USE GLOBAL, ALTERable GOTO, and CALL or CANCEL identifier.

Because nested programs cannot be called by programs outside the outermost program containing them, you only need to know their internal names when you are in DEBUG, debugging them or a program containing them (see Chapter 7 for their format).

The internal names of outermost programs are of the same format as external names.

Example

The following shows an outer program containing a nested program that itself contains a nested program. The example illustrates the external name of the outer program and the internal names of the nested program.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ANCESTOR-PROG.      The external name is ancestor_prog.
    :
IDENTIFICATION DIVISION.
PROGRAM-ID. CHILD-PROG.         The internal name is ancestor_prog$003$child_prog.
    :
IDENTIFICATION DIVISION.
PROGRAM-ID. GRANDCHILD-PROG.    The internal name is
    :                           ancestor_prog$004$grandchild_prog.
END PROGRAM GRANDCHILD-PROG.
END PROGRAM CHILD-PROG.
END PROGRAM ANCESTOR-PROG.
```

Chunk and Locality Set Names

The locality set name is extracted from the PROGRAM-ID paragraph, according to the external naming convention (refer to the *HP Link Editor/XL Reference Manual* for the definition of *locality set*). The executable code resides in the program file or an executable library. For a large program, the executable code is in chunks. Each chunk is in a separate subspace, but they all reside in the same program file and locality set.

The chunk name is the locality set name, with a three-digit number enclosed in dollar signs concatenated to it.

Example 1

If the *program-id* of a three-chunk program is `PROG-1`, then the locality set name is `prog_1`, and the chunk names are:

```
prog_1
prog_1$001$
prog_1$002$
```

You must be aware of chunks in these situations:

- When you read a verb map. Each code offset is offset from the beginning of a chunk. See Chapter 7 for information on maps.
- When you use DEBUG. Each code offset is offset from the beginning of a chunk. See Chapter 7 for information on debugging.
- When you use the LINK command. If you specify the MAP parameter, the command prints the code offset for each chunk. Refer to the *HP Link Editor/XL Reference Manual* for more information on the LINK command.

Example 2

The following program illustrates chunk names with nested and concatenated programs. The main program contains a subprogram, which contains another subprogram, and a concatenated program follows the main program's END PROGRAM header. All the programs are chunked. With nested or concatenated programs, a new chunk begins whenever a new program starts. Large programs may be chunked in the middle of the PROCEDURE DIVISION code.

```
IDENTIFICATION DIVISION.        Main program (first chunk) starts here.
PROGRAM-ID. MAIN-P.
PROCEDURE DIVISION.
    ⋮                           Second chunk starts here.
IDENTIFICATION DIVISION.        First subprogram (third chunk) starts here.
PROGRAM-ID. SUB-1.
PROCEDURE DIVISION.
    ⋮                           Fourth chunk starts here.
IDENTIFICATION DIVISION.        Second subprogram (fifth chunk) starts here.
PROGRAM-ID. SUB-2.
PROCEDURE DIVISION.
    ⋮                           Sixth chunk starts here.
END PROGRAM SUB-2.
END PROGRAM SUB-1.
END PROGRAM MAIN-P.

IDENTIFICATION DIVISION.        Concatenated subprogram (seventh chunk).
PROGRAM-ID. CONCAT-P.
PROCEDURE DIVISION.
     ⋮                          Eighth chunk starts here.
END PROGRAM CONCAT-P.
```

The following table gives the chunk location, number and name for the chunks in the above example:

Chunk Location	Chunk Number	Chunk Name
Main program	1	`main_p`
Main program	2	`main_p$001$`
Nested program	3	`main_p$003$sub_1`
Nested program	4	`main_p$003$sub_1$001$`
Nested program	5	`main_p$004$sub_2`
Nested program	6	`main_p$004$sub_2$001$`
Concatenated program	7	`concat_p`
Concatenated program	8	`concat_p$001$`

Data Alignment on MPE XL

By default, COBOL data is aligned in the following way on MPE XL:

- In the WORKING-STORAGE and FILE sections: level 01 and 77, COMP or BINARY SYNCHRONIZED, and index data items are 32-bit-aligned.
- In the LINKAGE section, all data items are 8-bit-aligned (byte-aligned), even if the SYNCHRONIZED clause is specified. The SYNCHRONIZED clause adds slack bytes so that every synchronized item is aligned on the same boundary as if the record were in WORKING-STORAGE.

The SYNC32 control option does not affect the above, because it specifies the default. The SYNC16 control option affects all sections. Its effect is the following:

- Level 01 and 77 data items are 32-bit-aligned.
- COMP or BINARY SYNCHRONIZED data items and indexed data items are 16-bit-aligned.

On MPE XL, the HP COBOL II/XL compiler assumes that all data items in the LINKAGE SECTION are byte-aligned unless the subprogram is compiled with the control option OPTFEATURES = LINKALIGNED or OPTFEATURES = LINKALIGNED16. In the first case, it assumes that they are word-aligned; in the second, halfword-aligned.

Parameter Checking

When your COBOL program calls an intrinsic, the compiler tries to match information about the type and alignment for each actual parameter. If this is not possible, the compiler issues an error message.

When your program calls a subprogram, the actual parameter list that your program passes is checked at link or load time against the formal parameter list of the subprogram for the following:

- The number of parameters.
- The parameter alignment.
- The types of parameters (if by value).

Each actual parameter must be of the same type as its corresponding formal parameter. The program file specifies the type of each formal parameter, and if the call is by value, the linker issues an error message if the corresponding actual parameter is not of that type.

All addresses on Series 900 machines are byte-addressed. An address that is divisible by two is *halfword-aligned*. An address that is divisible by four is *word-aligned* (and also *halfword-aligned*, since it is also divisible by two). An actual parameter and its formal counterpart must be aligned the same. Parameter alignment is checked at the following three times:

1. When the calling program is compiled.

   If the calling program is compiled with the control option OPTFEATURES = CALLALIGNED, the compiler issues an error message whenever a parameter in the CALL is not word-aligned.

   If the calling program is compiled with the control option OPTFEATURES = CALLALIGNED16, the compiler issues an error message whenever a parameter in the CALL is not halfword-aligned.

2. At link or load time.

   The linker or loader compares the alignments specified by the calling program and the called program. If alignments for the same parameter are different, the linker or loader issues an error message.

3. At execution time.

   If an actual and formal parameter are not aligned the same, this error is trapped (if possible) or the program results are unpredictable. The control option BOUNDS in the called program turns on the traps that detect this problem.

The number of parameters in the actual and formal parameter lists must be the same. The linker or loader issues an error message if they are not.

Parameter Passing

Your program can pass an actual parameter to a subprogram **by reference**, **by content**, or **by value**.

Passing Parameters by Reference

By default, your program passes an actual parameter by reference. This means that your program passes the address of the actual parameter to the subprogram. If the subprogram changes the value of its formal parameter, it also changes the value of your program's actual parameter.

Passing Parameters by Content

When your program passes an actual parameter by content, the compiler copies the actual parameter and passes the address of the copy to the subprogram. If the subprogram changes the value of its formal parameter, it changes the value of the copy, but it does not change the value of your program's actual parameter. Pass parameters by content to ensure that the subprogram does not change the value of data items.

Passing Parameters by Value

When your program passes an actual parameter by value, it only passes its value to the subprogram. If the subprogram changes the value of its formal parameter, it does not change the value of your program's actual parameter. To pass a parameter by value, enclose it in backslashes (\) in the CALL statement. Pass parameters by value to intrinsics and non-COBOL subprograms. Do not pass parameters by value to COBOL subprograms. Only numeric items can be passed by value.

Note Passing a parameter by content is not the same as passing it by value. The copy is passed by reference, and the called subprogram cannot tell that it has received a copy (the call is indistinguishable from a call by reference).

Parameter Alignment

On MPE XL, the HP COBOL II/XL compiler assumes that all data items in the LINKAGE SECTION are byte-aligned unless the subprogram is compiled with the control option OPTFEATURES=LINKALIGNED or OPTFEATURES=LINKALIGNED16. In the first case, it assumes that they are word-aligned. In the second case, it assumes that they are halfword-aligned.

If the subprogram is compiled with the control option BOUNDS, the compiler will trap parameter misalignment. Otherwise, you must be sure that your program's actual parameters are aligned the same or more restrictively as the subprogram's formal parameters. You can check the alignment of actual parameters by compiling the calling program with the control option OPTFEATURES=CALLALIGNED (or OPTFEATURES=CALLALIGNED16).

Passing and Retrieving a Return Value

You can return a value from a COBOL subprogram using the RETURN-CODE special register in the subprogram and the GIVING phrase in the calling program. The RETURN-CODE is used to pass a value back to the calling program.

Working with the Link Editor

The COBOL compiler does not inform the Link Editor of the formal parameter types for a COBOL subprogram. You must be sure that your program's actual parameters are of the same types and lengths as their formal counterparts.

The COBOL compiler generates an argument descriptor field for each parameter in the USING and GIVING clauses of the CALL statement. The Link Editor uses the argument descriptor fields to match and check parameters. You can specify the level of checking by specifying PARMCHECK in the Link Editor command LINK. Level of checking determines how many argument descriptor fields the Link Editor ignores.

Table 4-1 gives the values of the argument descriptor fields that the COBOL compiler sets for the Link Editor.

Table 4-1. Argument Descriptor Fields

Entity	Argument Descriptor Fields		
	Mode	Type	Alignment
Function	6 (function return)	Type that the function returns.	Alignment of function return type.
Reference parameter	2 (reference parameter)	Matches anything.	Alignment of type of parameter.
Value parameter	1 (value parameter)	Type of parameter.	Alignment of type of parameter.

Call Binding

The process of matching a subprogram call or an intrinsic call to its definition is called *call binding*. Call binding can occur at compile time, link time, load time, or execution time. Different subprograms called by the same program can be bound differently.

Subprogram Libraries

If your program contains a subprogram that other programs call, you may decide to put the subprogram in a library, making it available to other programs.

If you decide to put the subprogram in a library, you must decide what kind of library. The kinds of libraries are *relocatable library* (RL) and *executable library* (XL). In deciding, you should ask yourself:

1. How important is the size of the calling program?

   If you want the calling program to be as small as possible, you should use the executable library or XL.

2. How important is the execution speed of the calling program?

   If you want the calling program to execute as fast as possible, you should use the relocatable library or RL.

3. Will the subprogram need changes — fixes or updates — that are independent of the calling program?

   If the subprogram will need changes that are independent of the calling program, put it in an executable library. Then, when you change the subprogram, you will not have to relink every program that calls it.

It is recommended that you use the XL parameter (of the RUN command or Link Editor) to specify the name of the executable library that contains the subprogram. The alternative, which is only available if the executable library name is XL, is not portable (it is to let the RUN command reference the library with LIB=S (the default), LIB=P, or LIB=G).

Compile-Time Binding

Compile-time binding is performed by the compiler. It can only happen when the CALL or CANCEL target is a nested or concatenated program in the current source file.

The advantages of compile-time binding are:

- Calls to subprograms bound at compile time are faster than calls to subprograms bound at load time or execution time (but are the same speed as calls bound at link time).
- The program file is portable.

  The program file contains all the information that it needs to resolve calls to the subprogram. Other programmers can use your program without additional executable libraries.

The disadvantages of compile-time binding are:

- The program file is larger than it would be if it were bound at load time or execution time (but the same size as it would be if it were bound at link time).
- If you change any nested program, you must recompile and relink the program that contains it.

Terminology

The call rules that apply to subprograms bound at compile time require the introduction of new terminology, which is most easily explained with an example.

Example.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OUTER.              Outermost program.
PROCEDURE DIVISION.
BEGIN-OUTER.

IDENTIFICATION DIVISION.
PROGRAM-ID. MIDDLE.             Nested program.
PROCEDURE DIVISION.
BEGIN-MIDDLE.

IDENTIFICATION DIVISION.
PROGRAM-ID. INNER.              Nested program.
PROCEDURE DIVISION.
BEGIN-INNER.
END PROGRAM INNER.

END PROGRAM MIDDLE.

END PROGRAM OUTER.

IDENTIFICATION DIVISION.
PROGRAM-ID. NEXT.               Concatenated program.
PROCEDURE DIVISION.
BEGIN-NEXT.
END PROGRAM NEXT.
```

In the above program, OUTER is the *outermost program,* MIDDLE and INNER are *nested programs,* and NEXT is a *concatenated program.* OUTER *directly contains* MIDDLE, which directly contains INNER. OUTER *indirectly contains* INNER. NEXT, the concatenated program, is not contained in any other program, and is considered to be a *separately compiled program* (although it is in the same source file as OUTER).

Call Rules

The call rules that apply to subprograms bound at compile time are the following:

- Any program can call a concatenated program (except the program itself).
- Normally, a program can only call a nested program if it directly contains the nested program. However, if the nested program is COMMON, descendants of the program that contains the COMMON program can call it also. Only recursion is prohibited; that is, neither the COMMON program itself nor its descendants can call it.
- COMMON is valid only for nested programs.
- COMMON programs have access to all GLOBAL data that is valid at their nesting level (this applies to non-COMMON nested programs too).

Example. The following example illustrates the call rules.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. A.
 PROCEDURE DIVISION.
 BEGIN-A.
*Program A can call any program it directly contains: programs B and D.
*Program A cannot call any program it indirectly contains: program C.
 CALL "B".
 CALL "D".

 IDENTIFICATION DIVISION.
 PROGRAM-ID. B.
 PROCEDURE DIVISION.
 BEGIN-B.
*Program B can call any program it directly contains: program C.
*Program B can call any COMMON program directly contained by
*program A: program D.
 CALL "C".
 CALL "D".
```

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. C.
    PROCEDURE DIVISION.
    BEGIN-C.
   *Program C can call any COMMON program directly contained
   *by program A: program D.
    CALL "D".
    END PROGRAM C.

    END PROGRAM B.

   *Program D IS COMMON and is nested within program A.
    IDENTIFICATION DIVISION.
    PROGRAM-ID. D IS COMMON.
    PROCEDURE DIVISION.
    BEGIN-D.
    DISPLAY "SUCCESSFUL CALL TO COMMON PROGRAM".
    END PROGRAM D.

    END PROGRAM A.
```

Make a program COMMON when programs at various nesting levels need the function that it performs. For example:

- An application has a single program that reads and examines user data entered on-line. Many other routines call this program to read the next user input. Make the reading-and-examining program COMMON.
- A program handles a variety of error conditions that could happen anywhere in the execution flow. Make the error-handling program COMMON.

Link-Time Binding

Link-time binding is performed by the link editor. Subprogram code becomes part of the program file.

The advantages of link-time binding are:

- Calls to subprograms bound at link time are faster than calls to subprograms bound at load time or execution time.
- The program file is portable. The program file contains all the information that it needs to resolve calls to the subprogram. Other programmers can use your program without additional executable libraries.

The disadvantages of link-time binding are:

- The program file is larger.
- If you change the subprogram, you must relink all the programs that call it.

A subprogram can be input to the linker in either a relocatable object file or a relocatable library. The default is a relocatable object file, but the compiler can create a file of either type (see Chapter 6 for details on the RLFILE and RLINIT control options, which create relocatable libraries).

If only one program calls the subprogram, leave it in this relocatable object file. If more than one program calls the subprogram, or if the program that calls it also calls other subprograms, put it in a relocatable library.

Examples

This example uses a relocatable object file, SUBP, to hold the subprogram until it is linked with the main program. UPDPGM contains both the main program and the subprogram.

```
:COB85XL SUBPROG,SUBP;INFO="$CONTROL DYNAMIC"
:COB85XL MAINPROG,MAINP
:LINK FROM=MAINP,SUBP; TO=UPDPGM
```

The subprogram's relocatable object file can also be added to a specified relocatable library. The following adds a subprogram to the RL file named RLFILE:

```
:COB85XL SUBPROG,SUBP;INFO="$CONTROL SUBPROGRAM"
:LINKEDIT
BUILDRL RLFILE;LIMIT=10
ADDRL FROM=SUBP
EXIT
```

If you run this example, use the BUILDRL command only if an RL file does not already exist.

If a subprogram resides in a relocatable library, the subprogram can be linked with many different main programs. The subprograms are linked to the main program if it references the library (it need not specify the names of individual subprograms in the library). For more information on linking, see Chapter 6.

```
:LINK FROM=MAINP; TO=UPDPGM;RL=RLFILE
```

Load-Time Binding

Load-time binding is performed by the loader, which is invoked by the RUN command immediately before it executes the program. The subprogram must reside in an executable library (not in a relocatable object file or relocatable library). Subprogram code does not become part of the program file. Intrinsics are bound at load time.

The advantages of load-time binding are:

- Calls to subprograms bound at load time are thousands of times faster than calls to subprograms bound at execution time.
- You can update a subprogram in an executable library without having to relink the programs that call it. The more programs that call the subprogram, the more time saved.

The disadvantages of load-time binding are:

- Calls to subprograms bound at load time are slightly slower than calls to subprograms bound at link time (this only slows program execution significantly if the program calls the subprogram many times)
- To run your program, you (or another user) must have both the program file and the executable library.

Examples

The following commands create an executable library:

```
:LINKEDIT
BUILDXL XLFILE;LIMIT=10
EXIT
```

The following commands compile a subprogram and add the relocatable file to the executable library:

```
:COB85XL SUBPGM,SUBP;INFO="$CONTROL ANSISUB"
:LINKEDIT
ADDXL FROM=SUBP; TO=XLFILE; SHOW; MERGE
EXIT
```

Any main program calling this subprogram is compiled and linked separately. At execution time the executable library is searched, the externals are resolved, and the main program and subprogram are loaded for execution. The following commands accomplish these events:

```
:COB85XL MAINPGM,MAINP
:LINK FROM=MAINP; TO=UPDPGM
:RUN UPDPGM;XL="XLFILE"
```

Execution-Time Binding

Execution-time binding is performed by a calling program that contains a CALL *identifier* statement or an ON EXCEPTION or ON OVERFLOW phrase. The value of *identifier*, a subprogram name, is not available until execution time. At that time, the program calls a special routine that checks to see if the called program is contained in the calling program. If so, the subprogram is nested, and the special routine binds it to the calling program that contains it. If not, the subprogram is separately compiled, and the program calls HPGETPROCPLABEL, which binds the separately compiled subprogram. The subprogram must reside in an executable library specified by the XL=parameter of the link or run command, or in the program file.

The advantage of execution-time binding is:

- You specify the value of *identifier* and you can assign different subprogram names to *identifier* under different conditions.

The disadvantages of execution-time binding are:

- Calls to subprograms bound at execution time are thousands of times slower than calls to subprograms bound at compile time, link time, or load time.
- To run the program, you must have access to both the program file and the executable library that contains the subprogram.

The compiling and linking for the main program and subprogram is the same for load-time execution.

Examples

When coding the CALL statement in the main program, the *identifier* contains the subprogram name:

```
01 SUBP    PIC X (8).
   :
CALL SUBP USING VAR-A.
```

This allows the entry point to remain unresolved until execution time. During execution, the value of the field `SUBP` can be modified by the program based on input. When the CALL statement is executed, it issues a call for the value of `SUBP`. The following CALL statement calls the subprogram DATESUB1:

```
MOVE "DATESUB1" TO SUBP.
CALL SUBP USING VAR-A.
```

The subprograms must reside in the executable library specified by the XL=parameter of the link or run command, or in the program file.

Note Using the ON EXCEPTION or ON OVERFLOW phrase with the CALL *literal* statement slows the CALL statement by approximately .01 seconds and prevents the loader from catching missing subprograms. The ON EXCEPTION or ON OVERFLOW phrase defers checking for missing subprograms until execution time.

Switch Stubs

A switch stub is a program that allows your Native Mode HP COBOL II/XL program to call a subprogram compiled in Compatibility Mode. You do not have to change or recompile your program or the subprogram.

Figure 4-1 shows how a switch stub works. When the program calls the subprogram, what actually happens is that the program calls the switch stub and the switch stub calls the subprogram. This is transparent to the program and subprogram, except that performance is slower.

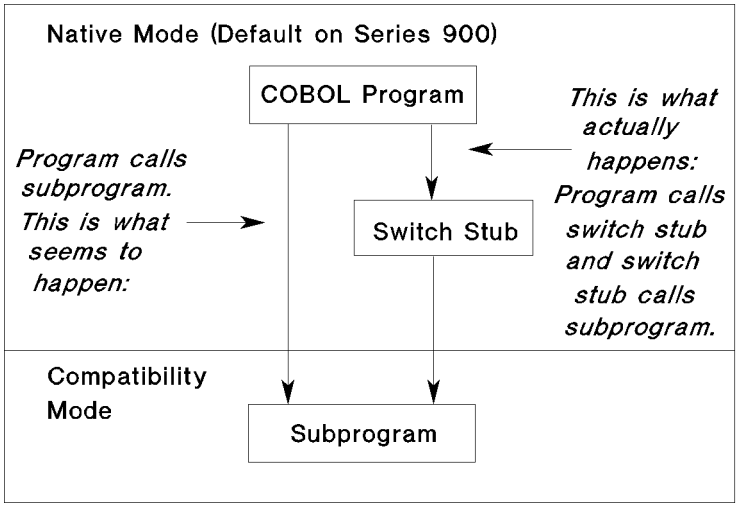


Figure 4-1. How a Switch Stub Works

You must write a switch stub for each Compatibility Mode subprogram that your program calls. You can create Pascal switch stubs using the Switch Assist Tool (SWAT). SWAT is an interactive utility. See "Calling Subprograms Written in SPL" for an example that uses SWAT. For complete information on SWAT, see the *Switch Programming User's Guide*.

Calling COBOL Subprograms

When your COBOL program calls a subprogram written in COBOL, you should understand the following:

- What type the subprogram is.
- How your program passes the actual parameters.
- Whether or not the subprogram uses the RETURN-CODE special register.
- When the calls to the COBOL subprogram will be bound to its definition.
- Whether to put the subprogram in a library, and if so, what kind of library.

Types of Subprograms

HP COBOL II/XL has three kinds of subprograms:

1. Non-Dynamic.
2. Dynamic.
3. ANSISUB.

You specify which kind of subprogram you want by using one of the following:

Table 4-2. Types of Subprograms and How to Specify Them

Subprogram Type	Option or Clause
Non-Dynamic.	$CONTROL SUBPROGRAM
Dynamic.	$CONTROL DYNAMIC or the INITIAL clause of the PROGRAM-ID paragraph.
ANSISUB	$CONTROL ANSISUB

For a description of these subprogram types, see the chapter "Interprogram Communication" in the *HP COBOL II/XL Reference Manual*.

When none of the subprogram options above is specified, the COBOL compiler uses the following rules to determine what type the program is:

- If the program is the first or only program in the source file, and the program contains a LINKAGE SECTION, it is assumed to be a non-dynamic subprogram.
- If the program is not the first program in the file (that is, if it is nested or concatenated) then it is assumed to be a non-dynamic subprogram.
- If the program is the first or only program in the file and it has no LINKAGE SECTION, it is assumed to be a main program.

Only the first program in the source file can be a main program. However, the first program is compiled as a subprogram if you specify any of the subprogram options listed in Table 4-2.

Table 4-3 compares some of the attributes of subprograms.

Table 4-3.
Comparison of Non-Dynamic, Dynamic, and ANSISUB Subprograms

Characteristic	Non-Dynamic	ANSISUB	Dynamic
Initialized:	Once, at link time.	Once, at link time.	Each time the subprogram is called.
Affected by the CANCEL statement?	No	Yes	No

Non-dynamic subprograms and ANSISUB subprograms are initialized once, at link time. Therefore, data in the WORKING-STORAGE SECTION retain their values between calls and files remain open between calls.

Dynamic subprograms are initialized every time the subprogram is called. Therefore, data in the WORKING-STORAGE SECTION do not retain their values between calls and files are closed between calls. Dynamic subprograms are not affected by the CANCEL statement. Therefore, they do not contain code to handle the CANCEL statement. Dynamic subprograms are smaller than ANSISUB subprograms.

If a program contains an INITIAL clause and the control option SUBPROGRAM or ANSISUB, the INITIAL clause takes precedence.

Calling Non-COBOL Subprograms

Your COBOL program can call subprograms and intrinsics written in C, FORTRAN 77, Pascal, and SPL. For each call to such a subprogram, the COBOL compiler generates the following information for the linker:

- The type of the identifier in the GIVING clause of the CALL statement.
- The alignment of each identifier in the USING clause of the CALL statement (for parameters that are passed by value, the type is generated also).
- The number of parameters.

You must ensure that the types of the actual parameters in the CALL statement are compatible with the types of their corresponding formal parameters in the non-COBOL subprogram. (Compile-time checking is impossible, because COBOL does not support forward declarations unless CALLINTRINSIC is used.)

The Control options CALLALIGNED and CALLALIGNED16 cause the compilers to issue a "Questionable" message for each parameter that is not aligned on a 32-bit or 16-bit boundary, respectively. The "Questionable" message applies to actual parameters.

Calling Subprograms Written in C

Your COBOL program can call a subprogram written in C if the parameters of the C routine are of types that have compatible COBOL types. If the C routine is a function, it must return a type that has a compatible COBOL type, and the *identifier* in the GIVING clause of the CALL statement must be of that compatible COBOL type.

Table 4-4. Compatible COBOL and C Types

COBOL Type	C Type
PIC X	char
PIC X(*n*)	char [*n*]
PIC S9 to S9(4)[1] USAGE COMP or BINARY level 01 or 77, or SYNC	short int
PIC S9(5) to S9(9)[1] USAGE COMP or BINARY level 01 or 77, or SYNC (but not $CONTROL SYNC16).	int

1 For best results, use the largest value in the range.

A C parameter is either passed by value or its address is passed by value. The latter is preceded by an asterisk (*) in the formal parameter declaration.

If the formal C parameter itself is passed by value, pass the actual COBOL parameter to it by value (enclose it in backslashes). If the address of the formal C parameter is passed by value, pass the actual COBOL parameter to it by reference.

Example

The following is a C subprogram that returns a nonvoid value:

```
# 1 "CGRANDE.PUBS.COBOL"
int grande (arr,len)
      int *arr;
      int       len;
{
      int largest = arr [0], i;
  for (i = 1; i< len; i++)
     if (largest < arr [i])
        largest = arr[i];
  return largest;
}
```

The following is a C subprogram that returns a void value:

```
# 1 "CREVERS.PUBS.COBOL"
void reverses (s)
       char   *s;
{
       int right = strlen (s) -1, left = 0;
       char t;
     for (; left < right; right--, left++) {
        t         = s [right];
        s [right] = s [left];
        s [left]  = t;
     }
}
```

Many C routines expect a null byte at the end of each ASCII string. The routine *reverses,* above, is an example of such a C routine. The following COBOL program, which calls *reverses,* declares a null-terminated ASCII string in COBOL. First it declares the null byte as a PIC X data item (which can be byte-aligned).

The following is a COBOL program that calls the two C subprograms above:

```
00001 COBCNTL  000010*  Defaults for Compatibility Mode compiler follow.
00002 COBCNTL  001100*CONTROL LIST,SOURCE,NOCODE,NOCROSSREF,ERRORS=100,NOVERBS,
               WARN
00003 COBCNTL  001200*CONTROL LINES=60,NOMAP,MIXED,QUOTE=",NOSTDWARN,SYNC16,
               INDEX16
00004 COBCNTL  001210*
00005 COBCNTL  001300*  Defaults for Native Mode compiler follow.
00006 COBCNTL  001400*
00007 COBCNTL  001600*CONTROL LIST,SOURCE,NOCODE,NOCROSSREF,ERRORS=100,NOVERBS,
               WARN
00008 COBCNTL  001700*CONTROL LINES=60,NOMAP,MIXED,QUOTE=",NOSTDWARN,SYNC32,
               INDEX32
00009 COBCNTL  001800*CONTROL NOVALIDATE,OPTIMIZE=0
00010 COBCNTL  001900*
00011 COBCNTL  002000*  For any other options, redirect COBCNTL.PUB.SYS using
00012 COBCNTL  002100*  a file equation.
00013 COBCNTL  002200*
00014          001000 ID DIVISION.
00015          002000 PROGRAM-ID.  CALLC.
00016          003000
00017          004000 DATA DIVISION.
00018          005000 WORKING-STORAGE SECTION.
00019          006000 01 TABLE-1.
00020          007000    05 TABLE-EL OCCURS 9  PIC S9(9) BINARY SYNC.
00021          008000 01 LARGST               PIC S9(9) BINARY SYNC.
00022          008100 01 STRING-REC.
00023          009000    05 STRING-1          PIC X(9) VALUE "ABCDEFGHI".
00024          009100    05 STRING-NULL       PIC X    VALUE LOW-VALUE.
00025          091100
00026          009200 01 LEN                  PIC S9(9) BINARY SYNC.
```

```
00027          011000
00028          012000 PROCEDURE DIVISION.
00029          013000 01-TEST.
00030          014000
00031          015000    MOVE 10 TO TABLE-EL(1).
00032          016000    MOVE  8 TO TABLE-EL(2).
00033          017000    MOVE 14 TO TABLE-EL(3).
00034          018000    MOVE  9 TO TABLE-EL(4).
00035          019000    MOVE 18 TO TABLE-EL(5).
00036          020000    MOVE 98 TO TABLE-EL(6).
00037          021000    MOVE  7 TO TABLE-EL(7).
00038          022000    MOVE 23 TO TABLE-EL(8).
00039          023000
00040          025100    MOVE 8 TO LEN.
00041          025200    CALL "GRANDE" USING TABLE-1 \LEN\ GIVING LARGST.
00042          026000    DISPLAY LARGST.
00043          027000
00044          028000
00045          029000
00046          030000    DISPLAY STRING-1 " BACKWARDS IS " WITH
                         NO ADVANCING
00047          031000    CALL "REVERSES" USING  STRING-1.
00048          032000    DISPLAY STRING-1.
```

The following are commands that compile, link, and execute the C subprograms and the COBOL program above:

```
:ccxl cgrande,cgrandeo,$null
:ccxl crevers,creverso,$null
:cob85xl callc,callco,$null
:link from=callco,cgrandeo,creverso;to=pcallc
:run pcallc
```

The COBOL program displays the following:

```
+000000098
ABCDEFGHI BACKWARDS IS IHGFEDCBA
```

Note The COBOL function REVERSE is more efficient than the above example in C.

Calling Subprograms Written in FORTRAN 77

Your COBOL program can call a subprogram written in FORTRAN 77 if the parameters of the FORTRAN 77 routine are of types that have compatible COBOL types.

Table 4-5 shows compatible COBOL and FORTRAN 77 types. FORTRAN 77 types that are not in Table 4-5 do not have compatible COBOL types. The number *n* is an integer.

Table 4-5. Compatible COBOL and FORTRAN Types

COBOL Type	FORTRAN 77 Type

Pass two variables to describe the FORTRAN string:
 	
`(1) A variable that is a group item or of`
`type PIC X, that contains the string.`
 	
`(2) A variable of type PIC S9(9) BINARY SYNC,`
`that contains the length of the string,`
`enclosed in backslashes.`
	`CHARACTER*`*n*

`PIC S9 to S9(4)`[1]
`USAGE COMP or BINARY.`
`Level 01 or 77, or SYNC.`
	`INTEGER*2`

`PIC S9(5) to S9(9)`[1]
`USAGE COMP or BINARY.`
`Level 01 or 77, or SYNC`
`(but not with $CONTROL SYNC16).`
	`INTEGER*4`

1 For best results, use the largest value in the range.

FORTRAN 77 integer parameters are always passed by reference. FORTRAN 77 character parameters are passed by descriptor (address, length, and value, in that order). If your COBOL program calls a FORTRAN 77 subprogram, it must pass the actual parameters by reference or by content.

Your COBOL program can call FORTRAN 77 functions of the types INTEGER*2 and INTEGER*4. The *identifier* in the GIVING clause of the CALL statement in your program must be of a type that is compatible with the type of the FORTRAN 77 function.

Your COBOL program cannot call FORTRAN 77 character functions, because the data name in the GIVING clause must be numeric.

Example

The following is a FORTRAN 77 subprogram:

```
      INTEGER*4 FUNCTION LARGER(A,L)
      INTEGER*4 A(8)
      INTEGER*4 LARGST,L
C
C         THIS SUBROUTINE FINDS THE LARGEST VALUE IN AN ARRAY
C         OF 'L' INTEGERS.
C
      LARGST = A(1)
      DO 100 I = 2,L
      IF (LARGST .LT. A(I)) LARGST = A(I)
 100  CONTINUE
      LARGER = LARGST
      RETURN
      END


C       ******************************************************
C       *               SUBROUTINE  BACKWRDS                 *
C       * THIS SUBROUTINE REVERSES AN ARRAY OF 'L' CHARACTERS*
C       ******************************************************

      SUBROUTINE BACKWRDS(STR)
      CHARACTER STR(10)
      CHARACTER N

      J = 10
      DO 100 K = 1,5
      N = STR(K)
      STR(K) = STR(J)
      STR(J) = N
      J = J - 1
 100  CONTINUE
      RETURN
      END
```

Note The COBOL function REVERSE is more efficient than the above example in FORTRAN.

The following COBOL program calls the FORTRAN 77 subprogram above:

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID.  CALLFTN.
003000 DATA DIVISION.
004000 WORKING-STORAGE SECTION.
005000 01 TABLE-INIT.
006000    05      PIC S9(9) COMP SYNC VALUE 10.
007000    05      PIC S9(9) COMP SYNC VALUE  8.
008000    05      PIC S9(9) COMP SYNC VALUE 14.
009000    05      PIC S9(9) COMP SYNC VALUE  9.
010000    05      PIC S9(9) COMP SYNC VALUE 18.
011000    05      PIC S9(9) COMP SYNC VALUE 98.
012000    05      PIC S9(9) COMP SYNC VALUE  7.
013000    05      PIC S9(9) COMP SYNC VALUE 23.
014000 01 TABLE-1 REDEFINES TABLE-INIT.
015000    05 TABLE-EL OCCURS 8
016000            PIC S9(9) COMP SYNC.
017000
018000 01 LARGEST-VALUE  PIC S9(9) COMP SYNC.
019000
020000 01 STRING-1 PIC X(10) VALUE "ABCDEFGHIJ".
021000 01 LEN     PIC S9(9) COMP SYNC.
022000
023000 PROCEDURE DIVISION.
024000 P1.
025000*****************************************************************
026000* Call FORTRAN 77 subroutine "LARGER" to find the largest      *
027000* element in a table on "LEN" elements.                        *
028000*****************************************************************
029000
030000      MOVE  8  TO  LEN.
031000      CALL "LARGER" USING TABLE-1, LEN GIVING LARGEST-VALUE.
032000      DISPLAY LARGEST-VALUE " IS THE LARGEST VALUE IN THE TABLE".
033000
034000*****************************************************************
035000* Call FORTRAN 77 subroutine "BACKWARDS" to reverse a string of*
036000* 10 characters.                                                *
037000* Shows passing character strings to  FORTRAN 77 subroutine    *
038000*****************************************************************
039000
040000      MOVE  10  TO  LEN.
041000      DISPLAY STRING-1 " BACKWARDS IS " WITH NO ADVANCING.
043000      DISPLAY STRING-1.
```

The following commands compile and link the FORTRAN 77 subprogram and the COBOL program:

```
:cob85xl callftn, callftno, $null
:ftnxl fortsub, fortsubo, $null
:link from=callftno,fortsubo;to=callftnp
```

The following command executes the COBOL program:

```
:callftnp
```

The COBOL program displays the following:

```
+000000098 IS THE LARGEST VALUE IN THE TABLE
ABCDEFGHIJ BACKWARDS IS JIHGFEDCBA
```

Calling Subprograms Written in Pascal

Your COBOL program can call a subprogram written in Pascal if the parameters of the Pascal subprogram are of types that have compatible COBOL types. Table 4-6 shows compatible COBOL and Pascal types, assuming default Pascal alignment. See the note below. Pascal types that are not in Table 4-6 do not have compatible COBOL types. The number *n* is an integer.

Table 4-6. Compatible COBOL and Pascal Types

COBOL Type	Pascal Type
PIC X	CHAR
PIC X(*n*)	PACKED ARRAY [*n*] OF where n is the length of the array.
PIC S9 to S9(4)[1] USAGE COMP or BINARY. Level 01 or 77, or SYNC.	SHORTINT
PIC S9(5) to S9(9)[1] USAGE COMP or BINARY. Level 01 or 77, or SYNC (but not with $CONTROL SYNC16)	INTEGER

1 For best results, use the largest value in the range.

Note You can specify any alignment for a Pascal type with the Pascal compiler option ALIGNMENT. Particularly, you can specify byte alignment (the COBOL default in the absence of SYNC) for all Pascal types. Refer to the *HP Pascal/XL Reference Manual* for more information on the ALIGNMENT compiler option.

Pascal VAR, UNCHECKABLE ANYVAR, and READONLY parameters are passed by reference. All other Pascal parameters are passed by value. If your COBOL program calls a Pascal subprogram, it must pass actual parameters to formal VAR parameters by reference, and pass numeric actual parameters to non-VAR parameters by value. (Parameters passed by value are enclosed in backslashes. For example: \X\).

Note Your COBOL program cannot call a Pascal program that has ANYVAR (as opposed to UNCHECKABLE ANYVAR) parameters, because each ANYVAR parameter has a hidden parameter that COBOL cannot detect.

Your COBOL program can call Pascal functions of types that have compatible COBOL types. The *identifier* in the GIVING clause of the CALL statement in your program must be of a type that is compatible with the type of the Pascal function.

Example

The following is a Pascal subprogram:

```
$SUBPROGRAM$
PROGRAM PASCSUB;
TYPE
   STRING_TYPE =  PACKED ARRAY[1..10] OF CHAR;
   ARRAY_TYPE  =  ARRAY[1..8] OF INTEGER;


              (* ************************************** *)
              (*        PROCEDURE    REVERSE            *)
              (* ************************************** *)

              (* THIS PROCEDURE WILL REVERSE A STRING OF   *)
              (* 'LEN' CHARACTERS.                         *)


PROCEDURE REVERSE(VAR STRING1 : STRING_TYPE;
                  LEN : INTEGER);
VAR
  J, K    : INTEGER;
  TEMP    : CHAR;
BEGIN
   J := LEN;
   FOR K := 1 TO LEN DIV 2  DO
   BEGIN
      TEMP       := STRING1[K];
      STRING1[K] := STRING1[J];
      STRING1[J] := TEMP;
              J  := J - 1;
   END;
END;
```

```
                (* ************************************* *)
                (*        PROCEDURE    GRANDE            *)
                (* ************************************* *)

                (* THIS PROCEDURE WILL FIND THE LARGEST      *)
                (* ITEM IN AN ARRAY OF 'L' ELEMENTS          *)


FUNCTION GRANDE( VAR ARR : ARRAY_TYPE;
                           L : INTEGER) : INTEGER;

VAR
  K : INTEGER;
  LARGEST : INTEGER;
BEGIN
   LARGEST := ARR[1];
   FOR K := 2 TO L  DO
     IF LARGEST < ARR[K]  THEN
        LARGEST := ARR[K];
   GRANDE := LARGEST;
END;

BEGIN
END.
```

Note The COBOL function REVERSE is more efficient than the above example in Pascal.

The following COBOL program calls the Pascal subprogram above:

```
001000 ID DIVISION.
002000 PROGRAM-ID.  IC807R.
003000
004000 DATA DIVISION.
005000 WORKING-STORAGE SECTION.
006000 01 TABLE-1.
007000    05 TABLE-EL OCCURS 8  PIC S9(9) BINARY SYNC.
008000 01 LARGST                PIC S9(9) BINARY SYNC.
009000 01 STRING-1              PIC X(10) VALUE "ABCDEFGHIJ".
010000 01 LEN                   PIC S9(9) BINARY SYNC.
011000
012000 PROCEDURE DIVISION.
013000 0001-TEST.
014000
015000      MOVE 10 TO TABLE-EL(1).
016000      MOVE  8 TO TABLE-EL(2).
017000      MOVE 14 TO TABLE-EL(3).
018000      MOVE  9 TO TABLE-EL(4).
019000      MOVE 18 TO TABLE-EL(5).
020000      MOVE 98 TO TABLE-EL(6).
021000      MOVE  7 TO TABLE-EL(7).
022000      MOVE 23 TO TABLE-EL(8).
023000
024000      MOVE 8 TO LEN.
025000      CALL "GRANDE" USING TABLE-1 \8\ GIVING LARGST.
026000      DISPLAY LARGST.
027000
028000
029000      MOVE 10 TO LEN.
030000      DISPLAY STRING-1 " BACKWARDS IS STRING " WITH NO ADVANCING.
031000      CALL "REVERSE" USING STRING-1 \10\.
032000      DISPLAY STRING-1.
```

The following commands compile, link, and execute the program and subprogram:

```
:pasxl pascsub,pascsubo,$null
:cob85xl callpas,callpaso,$null
:link from=callpaso,pascsubo;to=callpasp
:run callpasp
```

The COBOL program displays the following:

```
+000000098
ABCDEFGHIJ BACKWARDS IS STRING JIHGFEDCBA
```

Pascal and COBOL variables are both byte-aligned if:

- The Pascal subprogram includes the option $ALIGNMENT 1$.
- The COBOL program does not specify SYNC on elementary items in records.

If a COBOL program and a Pascal subprogram share a record that contains both 16- and 32-bit integers, the COBOL program must specify FILLER to ensure that COBOL aligns the 16-bit integers as Pascal does. Otherwise, COBOL aligns 16- and 32-bit integers on 32-bit boundaries (if SYNC is specified), while Pascal aligns 16-bit integers on 16-bit boundaries and 32-bit integers on 32-bit boundaries. ▮

Example

The following COBOL program passes a record to a Pascal procedure:

```
001000 ID DIVISION.
002000 PROGRAM-ID.  CPASREC.
003000
004000 DATA DIVISION.
005000 WORKING-STORAGE SECTION.
006000 01 PASCAL-RECORD.
007000    05 CHAR1                PIC X.
008000    05 FILLER               PIC X.
009000* INT-16-BITS is not synchronized because that would place it on
010000* a 32 bit boundary.  FILLER bytes were added to make sure that
011000* it is on a 16 bit boundary.
012000    05 INT-16-BITS          PIC S9(4) BINARY.
013000    05 INT-32-BITS          PIC S9(9) BINARY SYNC.
014000    05 STRING-VAR           PIC X(10).
015000
016000 PROCEDURE DIVISION.
017000 0001-TEST.
018000     CALL "DISPLAY-RECS" USING PASCAL-RECORD.
019000
020000     DISPLAY "A 1 BYTE CHAR, 16, AND 32 BIT INTEGERS"
021000             "  AND A STRING"
022000             " PASSED FROM A PASCAL RECORD".
023000     DISPLAY CHAR1.
024000     DISPLAY INT-16-BITS.
025000     DISPLAY INT-32-BITS.
026000     DISPLAY STRING-VAR.
```

The following Pascal program contains a procedure that the COBOL program calls:

```
$SUBPROGRAM$
PROGRAM PASREC;
TYPE
    PASCAL_RECORD = RECORD
                      CHAR1 : CHAR;
                      INT_16_BITS : SHORTINT;
                      INT_32_BITS : INTEGER;
                      STRING      : PACKED ARRAY [1..10] OF CHAR;
                    END;
PROCEDURE DISPLAY_RECS( VAR PREC : PASCAL_RECORD);
BEGIN
    WITH PREC DO
        BEGIN
            CHAR1 := 'A';
            INT_16_BITS := 9999;
            INT_32_BITS := -888888888;
            STRING := 'LMNOPQRSTU';
         END;
END;
BEGIN
END.
```

The following commands compile, link, and execute the COBOL and Pascal programs above:

```
:pasxl pasrec,pasreco,$null
:cob85xl cpasrec,cpasreco,$null
:link from=cpasreco,pasreco;to=cpasrecp
:run cpasrecp
```

The Pascal types *shortint* and *integer* are not byte-aligned by default (*shortint* is two-byte-aligned and *integer* is four-byte-aligned). The option $ALIGNMENT 1$ degrades performance.

Calculations with byte-aligned numbers do work in Pascal, but you can improve efficiency by assigning byte-aligned variables to temporary variables that have the default alignment and manipulating the temporary variables.

- Elementary items in the following COBOL record are byte-aligned.

COBOL record:

```
01 PACKED-REC.
   05 P1   PIC X(3).
   05 P2   PIC S9(4) COMP.
   05 P3   PIC X(2).
   05 P4   PIC S9(9) COMP.
```

To achieve byte-alignment in the corresponding Pascal record, use $ALIGNMENT 1$ as shown below:

In the Pascal record, the types *shortint_1* and *integer_1* are declared as follows:

```
shortint_1 = $ALIGNMENT 1$ shortint;
integer_1  = $ALIGNMENT 1$ integer;
```

Equivalent Pascal record:

```
packed_record = RECORD
   p1 : PACKED ARRAY [1..3] OF char;
   p2 : shortint_1;
   p3 : PACKED ARRAY [1..2] OF char;  {For PIC X(2)}
   p4 : integer_1;
END;
```

Calling Subprograms Written in SPL

Your COBOL program can call a subprogram written in SPL if the parameters of the SPL subprogram are of types that have compatible COBOL types. Table 4-7 shows compatible COBOL and SPL types. SPL types that are not in Table 4-7 do not have compatible COBOL types. The number *n* is an integer.

Table 4-7. Compatible COBOL and SPL Types

COBOL Type	SPL Type
PIC X(*n*)	BYTE ARRAY
PIC S9 to S9(4)[1] USAGE COMP or BINARY. Level 01 or 77, or SYNC.	INTEGER
PIC S9(5) to S9(9)[1] USAGE COMP or BINARY. Level 01 or 77, or SYNC	DOUBLE
PIC S9(10) to S9(18)[1] USAGE COMP or BINARY. SYNC.	Exact type is not available. Declare it as an INTEGER ARRAY (with four elements) in the SPL subprogram.
PIC S9(*n*) USAGE DISPLAY	Exact type is not available. Declare it as a BYTE ARRAY in the SPL subprogram.
PIC S9(*n*) USAGE COMP-3	Exact type is not available. Declare it as a BYTE ARRAY in the SPL subprogram.

1 For best results, use the largest value in the range.

Writing Switch Stubs

The SPL compiler runs only in Compatibility Mode, while the COBOL compiler runs in both Compatibility Mode and Native Mode. If your Native Mode COBOL program calls a Compatibility Mode SPL subprogram, then you must write a switch stub to switch from one mode to the other and pass parameters. The steps are:

1. Write the SPL and COBOL programs.
2. Use SWAT (Switch Assist Tool) to generate a Pascal switch stub. See "Switch Stubs" for more information.
3. Compile the SPL program.
4. Using the segmenter, put the SPL program USL into an SL.
5. Compile the COBOL program.
6. Compile the switch stub using the HP Pascal/XL compiler.
7. Link the COBOL program and the switch stub.
8. Execute the COBOL program.

Note When a Native Mode program calls a Compatibility Mode program, the Compatibility Mode program must be in an SL. When a Compatibility Mode program calls a Native Mode program, the Native Mode program must be in an executable library.

Example

The following illustrates the steps for a Native Mode COBOL program to call a Compatibility Mode SPL program.

Step 1. Write the SPL program:

```
$CONTROL SUBPROGRAM
BEGIN
<<*****************************************************************>>
<< CASCII                                                          >>
<<  Convert a number in base 2,8,10,16 to ascii string for double  >>
<< Parms:  DINT  is DOUBLE integer to convert  (VALUE)             >>
<<         BASE  is INTEGER one of 2,8,10,16   (VALUE)             >>
<<         STRING is BYTE ARRAY to convert into (REFERENCE)        >>
<< returns: NUMCHARS INTEGER is number of characters in string     >>
<<*****************************************************************>>
INTEGER PROCEDURE cascii( dint, base, string );
   VALUE dint,base;
   DOUBLE  dint;
   INTEGER base;
   BYTE ARRAY string;

   BEGIN
       INTEGER i;
       LOGICAL lint = dint + 1;      << for bit extraction >>
       BYTE ARRAY hexstring(0:15);
       INTRINSIC dascii;

       IF base = 8 OR base = 10 THEN BEGIN
           cascii := dascii( dint, base, string );
           IF base = 8 THEN cascii := 11;
           END
```

```
        ELSE IF base = 16 THEN BEGIN
          MOVE hexstring := "0123456789ABCDEF";
          FOR i := 7 STEP -1 UNTIL 0 DO BEGIN
            string(i):=hexstring(lint.(12:4));
            dint := dint & DLSR(4);
            END;
          cascii := 8;  << always hex string length >>
          END
        ELSE IF base = 2 THEN BEGIN
            FOR i := 31 STEP -1 UNTIL 0 DO BEGIN
                IF lint.(15:1) THEN string(i) := "1"
                               ELSE string(i) := "0";
                dint := dint & DLSR(1);
                END;
            cascii := 32;
            END
        ELSE cascii := 0;
        END;   << cascii >>
END.
```

Step 1 Continued. Write the COBOL program:

```
001000 ID DIVISION.
002000 PROGRAM-ID.  CALLSPL.
003000
004000 DATA DIVISION.
005000 WORKING-STORAGE SECTION.
006000 01 INTEGER              PIC S9(9) BINARY SYNC.
007000 01 BASE                 PIC S9(4) BINARY SYNC.
008000 01 STRING-VALUE         PIC X(32) VALUE SPACES.
009000 01 LEN                  PIC S9(4) BINARY.
010000 PROCEDURE DIVISION.
011000 0001-TEST.
012000      MOVE 123 TO INTEGER.
013000      MOVE 2 TO BASE.
014000      CALL "CASCII" USING \INTEGER\ \BASE\ @STRING-VALUE
015000           GIVING LEN.
016000      DISPLAY STRING-VALUE.
```

Step 2. Use SWAT to generate the switch stub:

To invoke SWAT, type the following command at the MPE XL prompt:

```
:SWAT
```

Screen 1. The first screen is the FILE screen. Type in the name of the file where you want the switch stub to go. In this case the file name is SWCASCII. Then press the (Enter) key.

```
HP30363A.01.00 Switch Assist Tool                                  FILE
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.


   [SWCASCII]   FILENAME


   SCRIPT PROCESSING ERRORS




  GOTO            ENTER   REFRESH                NEXT     HELP     EXIT
  MAIN             MODE    SCREEN
```

Figure 4-2. The FILE Screen

Screen 2. The next screen is the MAIN screen. Here you specify the name of the Compatibility Mode procedure and its parameters. Here these are `cascii`, `dint`, `base`, and `string`, respectively.

```
HP30363A.01.00 Switch Assist Tool                                  MAIN
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

       [swcascii]          NAME OF FILE TO HOLD GENERATED SOURCE CODE
       [cascii          ]  NAME OF TARGET CM PROCEDURE

P  #1  [dint                        ]  #2  [base                        ]
A      [string                      ]      [                            ]
R      [                            ]      [                            ]
A      [                            ]      [                            ]
M      [                            ]      [                            ]
A      [                            ]      [                            ]
T      [                            ]      [                            ]
E      [                            ]      [                            ]
R      [                            ]      [                            ]
       [                            ]      [                            ]
N      [                            ]      [                            ]
A      [                            ]      [                            ]
M      [                            ]      [                            ]
E      [                            ]      [                            ]
S      [                            ]      [                            ]
   #31 [                            ]  #32 [                            ]


  GOTO     GOTO    ENTER   REFRESH                 NEXT    HELP    EXIT
 FILFORM  COMMIT   MODE    SCREEN
```

Figure 4-3. The MAIN Screen

Screen 3. The next screen is the PROCINFO screen. Here you specify information about the Compatibility Mode procedure.

```
HP30363A.01.00 Switch Assist Tool                                  PROCINFO
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

       [cascii          ]  NAME OF TARGET ROUTINE

       LOCATION OF TARGET PROCEDURE     RETURN CONDITION CODE
       [x] GROUP SL                     [ ] YES, RETURN CONDITION CODE
       [ ] PUB SL                       [x] NO, DO NOT RETURN CODE
       [ ] SYSTEM SL

       FUNCTION RETURN TYPE
       [ ] NONE
       [ ] BYTE
       [x] INTEGER
       [ ] LOGICAL
       [ ] DOUBLE
       [ ] REAL
       [ ] LONG


  GOTO     GOTO     ENTER    REFRESH          PREV     NEXT     HELP     EXIT
  MAIN     COMMIT   MODE     SCREEN
```

Figure 4-4. The PROCINFO Screen

Screen 4. The next three screens are the PARMINFO screens where you specify information about each of the parameters of the Compatibility Mode procedure.

```
HP30363A.01.00 Switch Assist Tool                                  PARMINFO
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

       [DINT                          ]       PARAMETER NAME

       ADDRESSING METHOD              I/O TYPE
       [ ] REFERENCE                  [x] INPUT ONLY
       [x] VALUE                      [ ] OUTPUT ONLY
                                      [ ] INPUT/OUTPUT

       DATA TYPE                      ARRAY SPECIFICATION
       [ ] BYTE                       [x] NOT AN ARRAY
       [ ] INTEGER                    [ ] AN ARRAY
       [ ] LOGICAL
       [x] DOUBLE
       [ ] REAL
       [ ] LONG


  GOTO     GOTO     ENTER    REFRESH          PREV     NEXT     HELP     EXIT
  MAIN     COMMIT   MODE     SCREEN
```

Figure 4-5. The PARMINFO Screen for Parameter DINT

Screen 5.

```
HP30363A.01.00 Switch Assist Tool                                  PARMINFO
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

       [BASE                          ]       PARAMETER NAME

       ADDRESSING METHOD              I/O TYPE
       [ ] REFERENCE                  [x] INPUT ONLY
       [x] VALUE                      [ ] OUTPUT ONLY
                                      [ ] INPUT/OUTPUT

       DATA TYPE                      ARRAY SPECIFICATION
       [ ] BYTE                       [x] NOT AN ARRAY
       [x] INTEGER                    [ ] AN ARRAY
       [ ] LOGICAL
       [ ] DOUBLE
       [ ] REAL
       [ ] LONG


  GOTO     GOTO     ENTER    REFRESH        PREV     NEXT     HELP     EXIT
  MAIN     COMMIT   MODE     SCREEN
```

Figure 4-6. The PARMINFO Screen for Parameter BASE

Screen 6.

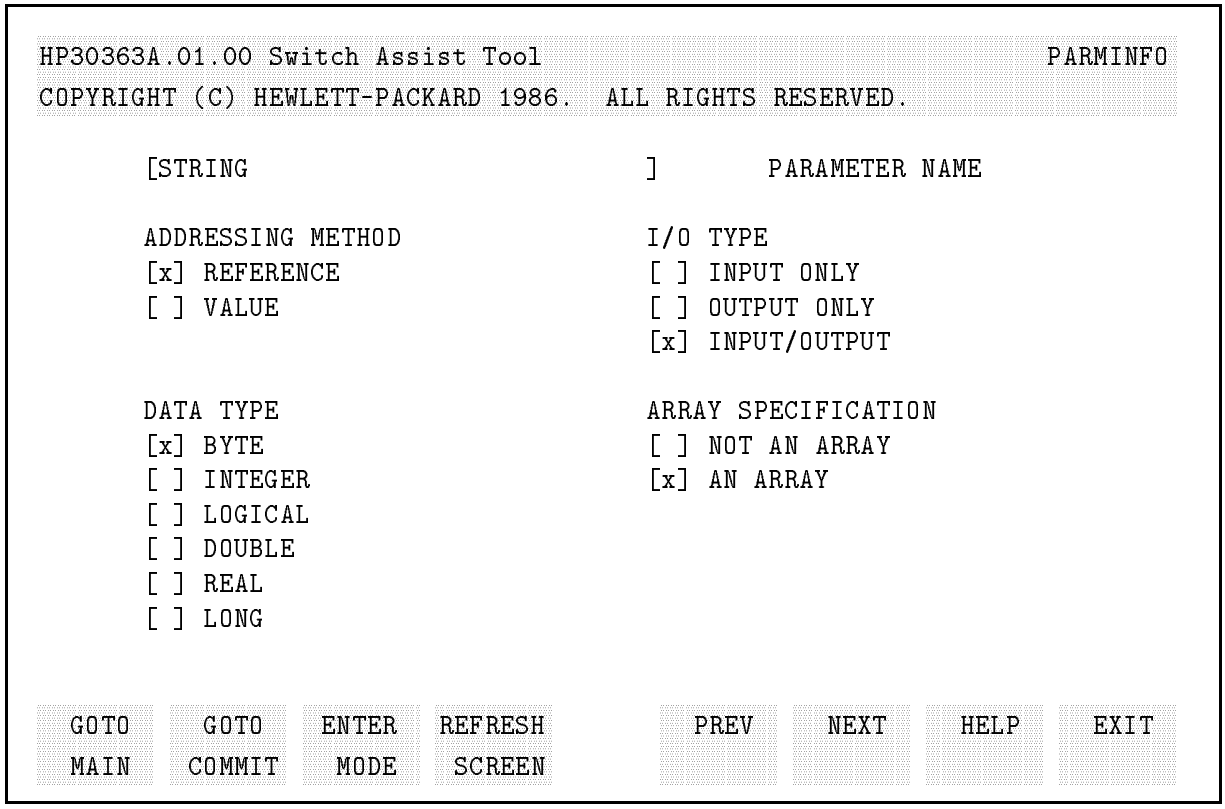


Figure 4-7. The PARMINFO Screen for Parameter STRING

Screen 7. The next screen is the ARRAYLEN screen where you specify information about the array parameter STRING.

```
HP30363A.01.00 Switch Assist Tool                                    ARRAYLEN
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

       [STRING                          ]       PARAMETER NAME

       LENGTH OF ARRAY
       [32   ] CONSTANT VALUE
       [                           ] NAME OF PARAMETER CONTAINING LENGTH

       ARRAY LENGTH USAGE
       [x] NUMBER OF ELEMENTS
       [ ] NUMBER OF BYTES
       [ ] NEGATIVE = BYTES / POSITIVE = ELEMENTS




 GOTO   GOTO   ENTER  REFRESH        PREV   NEXT   HELP   EXIT
 MAIN   COMMIT  MODE   SCREEN
```

Figure 4-8. The ARRAYLEN Screen for Parameter STRING

Screen 8. The final screen is the COMMIT screen where you start the code generation process.

```
HP30363A.01.00 Switch Assist Tool                                    COMMIT
COPYRIGHT (C) HEWLETT-PACKARD 1986.  ALL RIGHTS RESERVED.

   Press F2 when ready to begin generating code.









  GOTO    GENERATE   ENTER    REFRESH          PREV                HELP     EXIT
  MAIN    SOURCE     MODE     SCREEN
```

Figure 4-9. The COMMIT Screen

When you press softkey F2, the SWAT tool starts generating code and displays status messages. When SWAT is finished, it displays the FILE screen again. Press softkey F8 to exit.

This is the switch stub generated:

```
$check_actual_parm 0$
$check_formal_parm 0$
$os 'MPE/XL'$
$standard_level 'ext_modcal'$
$tables off$
$code_offsets off$
$xref off$
$type_coercion 'representation'$

{*****************************************************}
{*                                                   *}
{* Generated: THU, OCT  8, 1987,  5:46 PM            *}
{*                                                   *}
{* Switch Assist Tool HP30363A.00.00                 *}
{*                                                   *}
{*****************************************************}

PROGRAM Hp__stub_outer_block(input, output);

  CONST
    Hp__Pidt_Known           = 0;  { By number }
    Hp__Pidt_Name            = 1;  { By name   }
    Hp__Pidt_Plabel          = 2;  { By PLABEL }

    Hp__System_Sl            = 0;
    Hp__Logon_Pub_Sl         = 1;  { Logon PUB SL          }
    Hp__Logon_Group_Sl       = 2;  { Logon GROUP SL        }
    Hp__Pub_Sl               = 3;  { Program's PUB SL      }
    Hp__Group_Sl             = 4;  { Program's GROUP SL    }

    Hp__Method_Normal        = 0;  { Switch copy mode }
    Hp__Method_Split         = 1;
    Hp__Method_No_Copy       = 2;

    Hp__Parm_Value           = 0;  { value parameter }
    Hp__Parm_Word_Ref        = 1;  { reference parm, word addr }
    Hp__Parm_Byte_Ref        = 2;  { reference parm, byte addr }

    Hp__Ccg                  = 0;  { condition code greater (>) }
    Hp__Ccl                  = 1;  { condition code less    (<) }
    Hp__Cce                  = 2;  { condition code equal   (=) }
    Hp__All_Ok               = 0;  { Used in status check       }

  TYPE
    Hp__BIT8                 = 0..255;
    Hp__BIT16                = 0..65535;
    Hp__BIT8_A1              = $ALIGNMENT 1$ Hp__BIT8;
    Hp__BIT16_A1             = $ALIGNMENT 1$ Hp__BIT16;
    Hp__CM_PROC_NAME         = PACKED ARRAY [1..16] OF CHAR;
```

```
  Hp__GENERIC_BUFFER       = PACKED ARRAY [1..65535] OF CHAR;

  Hp__SCM_PROCEDURE        =
    PACKED RECORD
    CASE Hp__p_proc_id_type : Hp__BIT8 OF
      Hp__Pidt_Known:
        (Hp__p_fill        : Hp__BIT8_A1;
         Hp__p_proc_id     : Hp__BIT16_A1);
      Hp__Pidt_Name:
        (Hp__p_lib         : Hp__BIT8_A1;
         Hp__p_proc_name   : Hp__CM_PROC_NAME);
      Hp__Pidt_Plabel:
        (Hp__p_plabel      : Hp__BIT16_A1);
    END;  { record }

  Hp__SCM_IO_TYPE      = SET OF (Hp__input, Hp__output);

  Hp__PARM_DESC        =
    PACKED RECORD
    Hp__pd_parmptr         : GLOBALANYPTR;
    Hp__pd_parmlen         : Hp__BIT16;
    Hp__pd_parm_type       : Hp__BIT16;
    Hp__pd_io_type         : Hp__SCM_IO_TYPE;
    END;

  Hp__SCM_PARM_DESC_ARRAY  = ARRAY [0..31] OF Hp__PARM_DESC;

  HP__STATUS_TYPE      =
    RECORD
    CASE INTEGER OF
      0:
        (Hp__all       : INTEGER);
      1:
        (Hp__info      : SHORTINT;
         Hp__subsys    : SHORTINT);
    END;  { record }

{ Declare all types which can be passed to this stub    }
{ so that 16 bit alignments are allowed.                }

HP__SHORTINT                      = $ALIGNMENT 2$ SHORTINT;
HP__INTEGER                       = $ALIGNMENT 2$ INTEGER;
HP__REAL                          = $ALIGNMENT 2$ REAL;
HP__LONG                          = $ALIGNMENT 2$ LONGREAL;
HP__CHAR                          = $ALIGNMENT 1$ CHAR;

PROCEDURE HPSWITCHTOCM;
  INTRINSIC;

PROCEDURE HPSETCCODE;
  INTRINSIC;
```

```
  PROCEDURE QUIT;
    INTRINSIC;

{ End of OUTER BLOCK GLOBAL declarations }

$PAGE$

 FUNCTION CASCII $ALIAS 'CASCII'$
           (
          DINT : HP__INTEGER;
          BASE : HP__SHORTINT;
    ANYVAR STRING : Hp__GENERIC_BUFFER
           ) : HP__SHORTINT
        OPTION UNCHECKABLE_ANYVAR;

    VAR
      Hp__proc               : Hp__SCM_PROCEDURE;
      Hp__parms              : Hp__SCM_PARM_DESC_ARRAY;
      Hp__method             : INTEGER;
      Hp__nparms             : INTEGER;
      Hp__funclen            : INTEGER;
      Hp__funcptr            : INTEGER;
      Hp__byte_len_of_parm   : Hp__BIT16;
      Hp__cond_code          : SHORTINT;
      Hp__status             : HP__STATUS_TYPE;

 VAR Hp__retval : HP__SHORTINT;
 VAR Hp__loc_DINT : HP__INTEGER;
 VAR Hp__loc_BASE : HP__SHORTINT;

 begin { STUB procedure CASCII }


{*****************************************************}
{*                                                   *}
{* Generated: THU, OCT  8, 1987,  5:46 PM            *}
{*                                                   *}
{* Switch Assist Tool HP30363A.00.00                 *}
{*                                                   *}
{*****************************************************}


  { Initialization }


  { Setup procedure information--name, lib, etc. }

  Hp__proc.Hp__p_proc_id_type :=  Hp__Pidt_Name; { By name }
  Hp__proc.Hp__p_lib          :=  Hp__Group_Sl;
  Hp__proc.Hp__p_proc_name    :=  'CASCII          ';
```

```
{ Setup misc. variables }

Hp__method        :=  Hp__Method_Normal;
Hp__nparms        :=  3;

{ Setup length/pointers for functional return if this }
{ is a FUNCTION.  Set length to zero, pointer to NIL  }
{ if this is not a FUNCTION.                          }
Hp__funclen       :=  SIZEOF(Hp__retval);
Hp__funcptr       :=  INTEGER(LOCALANYPTR(ADDR(Hp__retval)));

{ Make a local copy of all VALUE parameters }

Hp__loc_DINT := DINT;
Hp__loc_BASE := BASE;

{ Build parameter descriptor array to describe each }
{ parameter.                                        }


{ DINT -- Input Only by VALUE }

Hp__byte_len_of_parm := 4;

Hp__parms[0].Hp__pd_parmptr   :=
             ADDR(Hp__loc_DINT);
Hp__parms[0].Hp__pd_parmlen   := Hp__byte_len_of_parm;
Hp__parms[0].Hp__pd_parm_type := Hp__Parm_Value;
Hp__parms[0].Hp__pd_io_type   := [Hp__input];

{ BASE -- Input Only by VALUE }

Hp__byte_len_of_parm := 2;

Hp__parms[1].Hp__pd_parmptr   :=
             ADDR(Hp__loc_BASE);
Hp__parms[1].Hp__pd_parmlen   := Hp__byte_len_of_parm;
Hp__parms[1].Hp__pd_parm_type := Hp__Parm_Value;
Hp__parms[1].Hp__pd_io_type   := [Hp__input];

{ STRING -- Input/Output by REFERENCE }

Hp__byte_len_of_parm := 32;

Hp__parms[2].Hp__pd_parmptr   :=
             ADDR(STRING);
Hp__parms[2].Hp__pd_parmlen   := Hp__byte_len_of_parm;
Hp__parms[2].Hp__pd_parm_type := Hp__Parm_Byte_Ref;
Hp__parms[2].Hp__pd_io_type   := [Hp__input, Hp__output];
```

```
{ Do the actual SWITCH call }

HPSWITCHTOCM(Hp__proc,         { Procedure info          }
             Hp__method,       { Switch copy method      }
             Hp__nparms,       { Number of parameters    }
             Hp__parms,        { Parm descriptor array   }
             Hp__funclen,      { func ret value length   }
             Hp__funcptr,      { Addr of func return     }
             Hp__cond_code,    { cond. code return       }
             Hp__status);      { SWITCH status code      }

if (Hp__status.Hp__all  Hp__all_ok) then
  BEGIN { SWITCH subsystem error }
  QUIT(Hp__status.Hp__info);
  END;  { SWITCH subsystem error }

CASCII := Hp__retval;

end; { STUB procedure }

BEGIN { Program Outer block code }

END.  { Program Outer block code }
```

Step 3. Compile the SPL program.

```
:spl cascii,casciio,$null
```

Step 4. Using the segmenter, put the SPL program USL in an SL:

```
:segmenter

HP32050A.02.00  SEGMENTER/3000 (C) HEWLETT-PACKARD CO 1985
-buildsl sl,300,1
-usl casciio
-listusl

USL FILE CASCII0.PUBS.COBOL74

SEG'
   CASCII            152 P   A C N R

FILE SIZE     144000(  620.  0)
DIR. USED        235(    1. 35)    INFO USED        161(    0.161)
DIR. GARB.         0(    0.  0)    INFO GARB.         0(    0.  0)
DIR. AVAIL.    14143(   60.143)    INFO AVAIL.   127217(  535. 17)

-addsl seg'
-listsl

SL FILE SL.PUBS.COBOL74

SEGMENT   0 SEG'            LENGTH   160

   ENTRY POINTS    CHECK CAL STT  ADR
   CASCII            0    C    1     0

   EXTERNALS       CHECK STT SEG
   DASCII            0     2   ?

1

USED            1600(    7.  0)    AVAILABLE     111200(  445.  0)

-exit

END OF SUBSYSTEM
```

Step 5. Compile the COBOL and switch stub programs:

```
:cob85xl callspl,callsplo,$null
:pasxl swcascii,swcascio,$null
```

Step 6. Link the COBOL program and the switch stub:

```
:link from=callsplo,swcascio;to=callsplp
```

Step 7. Execute the COBOL program:

```
:run callsplp
```

EXTERNAL Data Items and Files

EXTERNAL data items and files can be shared by two or more programs (for the purpose of this discussion, a subprogram is also a program). For programs to share an EXTERNAL item:

- Each program must declare the EXTERNAL item (data item or file).
- Each program must give the EXTERNAL item exactly the same name, because the Link Editor matches it to itself in all programs by its external name (see "External Naming Convention").
- Programs that share the EXTERNAL item must either be linked together or reside in the same object module in an executable library. (To put programs in the same object module in an executable library, use the MERGE option of the Link Editor command ADDXL).

In the case of shared EXTERNAL records, there is an easy way to ensure that their names and the names of the items within them are exactly the same in the programs that share them: put their declarations in a COPY library and copy the library into each program with the COPY statement.

For an EXTERNAL record, the compiler generates one external name. To find the actual location of an EXTERNAL record, consult the Link Map (see Chapter 7).

The compiler generates two external names for an EXTERNAL file—one for the FD name (based on the file name) and one for the record area (which is the file name with "_buffer_" appended to it).

Uses for EXTERNAL data items and files are:

- To allow a main program and separately compiled subprograms to share files and data (nested programs can share files and data using the GLOBAL clause).
- To pass parameters between programs without the USING phrase.

  Programs that change data other than that passed through the USING phrase have side effects, though, so be very careful.
- To reduce program file size.

  EXTERNAL items are stored in space allocated at run time, while internal items are stored in the program file. Therefore, you can significantly reduce program file size by declaring records that contain huge arrays EXTERNAL.

Note If a file is declared EXTERNAL in one program, and is opened in another program that uses the EXCLUSIVE statement, then the first program must declare the file with L in the ASSIGN clause. This enables dynamic locking.

Similarly, if a file is declared EXTERNAL in a program that does not write to it, it must declare it with CCTL in the ASSIGN clause to enable CCTL.

Example

This example shows how to invoke CONTROL Y traps from COBOL. The main program executes a loop until CONTROL Y is pressed. The subprogram arms the CONTROL Y trap to execute its secondary entry point when CONTROL Y is pressed. The main program and subprogram communicate through EXTERNAL data items.

The following is the main program:

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID. CONTROL-Y-TEST.
001200 ENVIRONMENT DIVISION.
001300 CONFIGURATION SECTION.
001400 SPECIAL-NAMES.
001600     SYMBOLIC CHARACTERS BELL IS 8.
001700 DATA DIVISION.
001800 WORKING-STORAGE SECTION.
001900 1 TOTAL          PIC S9(9) COMP VALUE 0.
002100 1 CONTROL-Y      EXTERNAL PIC X.
002200   88 CONTROL-Y-HIT VALUE "Y".
002250   88 CONTROL-Y-OFF VALUE "N".
002600 1 LOTS-OF-STUFF  EXTERNAL.
002700   5              PIC X(40).
002300 PROCEDURE DIVISION.
002350 P1.
002400     SET CONTROL-Y-OFF TO TRUE.
002500     MOVE ALL "*" TO LOTS-OF-STUFF.
002600     CALL "ARM-CONTROL-Y".
002700 LOOP.
002800     DISPLAY "HI" WITH NO ADVANCING.
002900     ADD 1 TO TOTAL.
003000     IF NOT CONTROL-Y-HIT GO LOOP.
003100*
003200     DISPLAY BELL.
003300     DISPLAY "control-y was hit after " TOTAL " times".
003400     DISPLAY LOTS-OF-STUFF.
003500     SET CONTROL-Y-OFF TO TRUE.
003600     MOVE 0 TO TOTAL.
003700     GO TO LOOP.
```

The following is the subprogram:

```
000100$CONTROL DYNAMIC
001200 IDENTIFICATION DIVISION.
001300 PROGRAM-ID. ARM-CONTROL-Y.
001800 DATA DIVISION.
001900 WORKING-STORAGE SECTION.
002000 1 TOTAL          PIC S9(9) COMP VALUE 0.
002100 1 PROCNAME       PIC X(20) VALUE "!control_y_trap!".
002200 1 PLABEL         PIC S9(9) COMP.
002300 1 OLDPLABEL      PIC S9(9) COMP.
002400 1 PROGFILE       PIC X(40).
002500*    another way to pass data
002510 1 CONTROL-Y      EXTERNAL PIC X.
002520   88 CONTROL-Y-HIT VALUE "Y".
002530   88 CONTROL-Y-OFF VALUE "N".
002600 1 LOTS-OF-STUFF  EXTERNAL.
002700   5              PIC X(40).
002800 PROCEDURE DIVISION.
002850 P1.
002900*    get plabel from hpgetprocplabel
003000     CALL INTRINSIC "HPMYPROGRAM" USING PROGFILE.
003100     CALL INTRINSIC "HPGETPROCPLABEL" USING PROCNAME PLABEL \\
003200                                            PROGFILE.
003300*    call xcontrap
003400     CALL INTRINSIC "XCONTRAP" USING PLABEL OLDPLABEL.
003500     EXIT PROGRAM.
003600*
003700 ENTRY "CONTROL-Y-TRAP".
003800     SET CONTROL-Y-HIT  TO TRUE.
003900     MOVE "Trap routine reenabled again" TO LOTS-OF-STUFF.
004000*    reenable for next time
004100     CALL INTRINSIC "RESETCONTROL".
```

EXTERNAL Items and FORTRAN

The Link Editor matches FORTRAN named common blocks and COBOL EXTERNAL records by name (one FORTRAN named common block matches one COBOL EXTERNAL record).

If the FORTRAN named common block declares more than one variable, it is your responsibility to align the elementary items of the COBOL record along the proper boundaries. You may have to specify unused bytes with FILLER. Remember that COBOL aligns all USAGE BINARY SYNCHRONIZED data items along 32-bit boundaries, regardless of data item size (unless you compile the program with the SYNC16 control option, in which case these data items are 16-bit-aligned).

EXTERNAL Items and Pascal

The Link Editor matches Pascal variables declared in the outer block and COBOL EXTERNAL records. The Pascal program must be compiled with the EXTERNAL, GLOBAL, or GLOBAL and EXTERNAL options.

Every variable in the outer block of the Pascal program must have a matching COBOL EXTERNAL record if the Pascal EXTERNAL option is used. This applies only between Pascal and COBOL.

EXTERNAL Items and C

The Link Editor matches C global variables and COBOL EXTERNAL records.

Sharing EXTERNAL Items

When two COBOL programs share EXTERNAL items, one program can declare some EXTERNAL items that the other program does not. When a COBOL and a FORTRAN (or C) program share EXTERNAL items, the COBOL program can declare EXTERNAL records that do not correspond to FORTRAN named common blocks (or C globals) or vice versa.

COBOL, FORTRAN, and Pascal Example

This COBOL main program, FORTRAN subprogram, and Pascal subprogram can pass information to each other through shared EXTERNAL items.

The following is the COBOL main program with EXTERNAL records:

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID. COBEXT.
001200 DATA DIVISION.
001300 WORKING-STORAGE SECTION.
001400 01 A EXTERNAL.
001500   05  I               PIC S9(9) BINARY.
001600   05  J               PIC S9(4) BINARY.
001610   05  FILLER          PIC XX.
001620   05  K               PIC S9(9) BINARY.
001700 PROCEDURE DIVISION.
001800 P1.
001900     CALL "PAS".
002000     MOVE -7 TO I.
002000     MOVE -8 TO J.
002000     MOVE -9 TO K.
002100     CALL "FTN".
002200     DISPLAY "I=" I ", J=" J ", K=" K.
```

The following is the FORTRAN subprogram with named common block:

```
SUBROUTINE FTN
COMMON /A/ I,J,K
INTEGER*4 I,K
INTEGER*2 J
WRITE(6,*) I,J,K
END
```

The following is the Pascal subprogram. It is compiled with the EXTERNAL option:

```
$EXTERNAL,SUBPROGRAM$
PROGRAM STUFF;
TYPE
     COM = RECORD
           I: INTEGER;
           J: SHORTINT;
           K: INTEGER;
           END;
VAR
     A : COM;

PROCEDURE FTN; EXTERNAL FTN77;

PROCEDURE PAS;
   BEGIN
   A.I:=5;
   A.J:=-5;
   A.K:=6;
   FTN;
   END;
BEGIN
END.
```

GLOBAL Data Items and Files

GLOBAL data items and files can be shared by two or more programs (or subprograms). For programs to share a GLOBAL item:

- One program must declare the GLOBAL item (data item or file).
- The other programs must be nested within the program that declares the GLOBAL item.
- A program nested within the program that declares the GLOBAL item cannot have a local item with the same name and qualification as the GLOBAL item. If it does, references to that name refer to the local item rather than the GLOBAL item.

For more information, see Chapter 3.

Note

If a file is declared GLOBAL in one program, and is opened in another program that uses the EXCLUSIVE statement, then the first program must declare the file with L in the ASSIGN clause. This enables dynamic locking.

Similarly, if a file is declared GLOBAL in a program that does not write to it, it must declare it with CCTL in the ASSIGN clause to enable CCTL.

Calling Intrinsics

The section explains some aspects of calling intrinsics. For more information about HP Intrinsics, see the *MPE XL Intrinsics Reference Manual*.

Using $CONTROL CALLINTRINSIC

Intrinsics are subprograms whose declarations reside in the intrinsic file, SYSINTR.PUB.SYS. Ideally, your program always calls an intrinsic with CALL INTRINSIC instead of CALL, and does not use the control option CALLINTRINSIC.

The control option CALLINTRINSIC increases compile time because it causes the compiler to search SYSINTR.PUB.SYS every time your program calls a subprogram with CALL *literal*. If the subprogram declaration is in SYSINTR.PUB.SYS, the compiler assumes that you intended to call it as an intrinsic, and the compiler does the following:

- Issues a warning.
- Generates code for the subprogram as if your program had called it as an intrinsic.

If you are not sure whether your program always calls an intrinsic with CALL INTRINSIC, compile it with the CALLINTRINSIC control option, which will identify all of the intrinsic calls that use CALL. In those calls, change CALL to CALL INTRINSIC. Then, recompile your program without the CALLINTRINSIC control option.

How Intrinsics Are Called

When your program calls a subprogram as an intrinsic, the compiler reads from SYSINTR.PUB.SYS the following information about each formal subprogram parameter:

- Whether the formal paramater is passed by reference or value.

  Don't explicitly specify reference or value. For example, do not include the character \ to pass a data item by value, or the @ sign to pass a data item by reference.

- The default value of the formal parameter.

  This value is assigned to the formal parameter if your program does not provide an actual parameter for it. Specify \\ for each parameter omitted.

- The type of the formal parameter.

  The compiler issues an error message if the types of the formal and actual parameters are not compatible. See Table 4-8 for a list of intrinsic parameter types and corresponding COBOL types.

- The alignment of the formal parameter.

  The compiler issues an error message if the actual parameter does not have the same alignment as the formal parameter.

Table 4-8 lists the intrinsic parameter types and their corresponding COBOL types.

Table 4-8. Intrinsic Parameter Types and Corresponding COBOL Types

Mnemonic	Full Name of Intrinsic Parameter Type	Size in Bytes	COBOL Type (Passed by Reference)	COBOL Type (Passed by Value)
A	Array	*n*	USAGE DISPLAY USAGE PACKED_DECIMAL Group item	Numeric items[1] Nonnumeric items[1]
B	Boolean	1	Group item	Numeric items Nonnumeric items[1]
C	Character	1	USAGE DISPLAY Group item	Numeric items Nonnumeric items[1]
-	Packed decimal	*n*	USAGE PACKED-DECIMAL	Not applicable
-	Entry point	4	S9(9) BINARY	Not compatible
-	External ASCII	*n*	USAGE DISPLAY Group item	Not applicable
I16	16-bit signed integer	2	S9(1)-S9(4) BINARY	Numeric items
I32	32-bit signed integer	4	S9(5)-S9(9) BINARY	Numeric items
I64	64-bit signed integer	8	S9(10)-S9(18) BINARY	Numeric items
-	Procedure	0	Not compatible	Not compatible
REC	Record (generic structure)	*n*	USAGE DISPLAY USAGE PACKED-DECIMAL Group item	Numeric items[1] Nonnumeric items[1]
R32	32-bit real	4	Not compatible	Numeric items[2]
R64	64-bit real	8	Not compatible	Numeric items[2]
R128	128-bit real	16	Not compatible	Not compatible
U16	16-bit unsigned integer	2	S9(1)-S9(4) BINARY	Numeric items
U32	32-bit unsigned integer	4	S9(9) BINARY	Numeric items
U64	64-bit unsigned integer	8	S9(18) BINARY	Numeric items
-	Constant address	4	Not compatible	Not compatible
-	Local label address	4	Not compatible	Not compatible
@32	32-bit address	4	S9(9) BINARY	Not compatible
@64	64-bit address	8	S9(18) BINARY	Not compatible
S	Set	4	Not compatible	Numeric items[1]

1 Size must match exactly. No type conversion is done.

2 For R32, use S9(9) BINARY. For R64, use S9(18) BINARY.

For BINARY fields, an intrinsic may return a value outside the range of valid numbers for the COBOL type. Calculations with such values may cause a SIZE ERROR. To prevent the error, use a MOVE statement to move the contents of such fields to larger BINARY fields. For example, the WHO intrinsic may return 16385 for the terminal parameter. Because the terminal parameter is COBOL type S9(4) BINARY, the value of terminal can be moved to a COBOL type S9(5) BINARY to match the type with the contents.

Passing Real Numbers to Intrinsics

Some intrinsics have parameters that are real numbers, for example PAUSE. You can call these intrinsics by converting a numeric character string representing the real number into floating-point format. The intrinsic HPEXTIN converts a numeric character string into a floating-point value. For more information about HPEXTIN, see the *Compiler Library/XL Reference Manual*.

Example

The following program reads a numeric value from the terminal, converts the value to floating-point, and passes it to the PAUSE intrinsic.

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID.  COBPAUSE.
001200 DATA DIVISION.
001300 WORKING-STORAGE SECTION.
001400 01  CHAR-STRING     PIC S999 SIGN IS LEADING SEPARATE.
001500 01  STRING-LEN      PIC S9(4) BINARY.
001600 01  REAL-SECONDS    PIC S9(9) BINARY VALUE ZERO.
001700 01  ERROR-CODE      PIC S9(4) BINARY VALUE ZERO.
001800 PROCEDURE DIVISION.
001900 FIRST-PARA.
002000    DISPLAY "Enter number of seconds to pause."
002100    ACCEPT CHAR-STRING FREE
002150    CALL INTRINSIC ".LEN." USING CHAR-STRING GIVING STRING-LEN
002200    CALL INTRINSIC "HPEXTIN" USING CHAR-STRING STRING-LEN
002300                      0 1 0 0 REAL-SECONDS ERROR-CODE
002400    IF ERROR-CODE <> 0
002500        PERFORM HPEXTIN-ERROR
002600    ELSE
002700        CALL INTRINSIC "PAUSE" USING REAL-SECONDS
002800    END-IF
002900    STOP RUN.
003000
003100 HPEXTIN-ERROR.
003200    DISPLAY "HPEXTIN ERROR = " ERROR-CODE.
```

5

Files

Introduction

Files are the basis for input and output. Your COBOL program reads input from files and writes output to them.

Files are also a means of interprogram communication. Two or more programs can communicate using a shared file. See "DATA DIVISION: GLOBAL Data Items and Files" in Chapter 3 and "EXTERNAL Data Items and Files" in Chapter 4.

Files that your program declares are called *logical files*. Files that exist outside your program are called *physical files*. When you associate a logical file with a physical file, everything that your program does to the logical file happens to the physical file. Before your program can access a logical file and its associated physical file, the program must open the file with the OPEN verb.

Note You should not use intrinsics to access a file opened by the COBOL OPEN statement. When you OPEN a file, the HP COBOL II/XL run-time system assumes all accesses to that file are done with COBOL statements. If you use intrinsics to access the file before closing it with the CLOSE statement, the results are unpredictable.

This chapter presents the following:

- Lists the input and output statements and which can be used with each of the four types of logical files.
- Explains the four types of logical files that your program can use.
- Explains how you can use variable length records.
- Explains how physical files are created and associated with logical files.
- Gives information on overwriting, updating, and appending to files.
- Gives the status codes for file errors.

Table 5-1 lists each I-O statement and how each statement is used with each file type. The following explains what each entry in Table 5-1 means:

Meanings of Entries in Table 5-1

Table Entry	Meaning
Comment	This verb or clause is treated as a comment.
Illegal	This verb or clause is illegal for this file type.
Optional	This verb or clause is optional. Use it if you want the functionality it provides.
Required	This verb or clause is required.
Select one	Select any one of the items in the box with this entry.
Blank	This verb or clause is not applicable to this file type.

Table 5-1. I-O Statements and File Types

I-O Statement		How the Statement Can Be Used			
Verb	Clause/Phrase	Sequential Organization File	Relative Organization File	Indexed Organization File	Random Access File
CLOSE					
	REEL FOR REMOVAL	Comment[1]	Illegal	Illegal	Illegal
	UNIT FOR REMOVAL	Comment[1]	Illegal	Illegal	Illegal
	WITH NO REWIND	Optional	Illegal	Illegal	Illegal
	WITH LOCK	Optional	Optional	Optional	Optional
FD		Required	Required	Required	Required
	BLOCK	Optional	Optional	Optional	Optional
	RECORDING MODE	Optional	Optional	Optional	Optional
	RECORD CONTAINS	Optional	Optional	Optional	Optional
	LABEL RECORDS	Optional	Optional	Optional	Optional
	VALUE OF	Optional	Optional	Optional	Optional
	DATA RECORDS	Comment	Comment	Comment	Comment
	LINAGE	Optional	Illegal	Illegal	Illegal
	WITH FOOTING	Optional			
	LINES AT TOP	Optional			
	LINES AT BOTTOM	Optional			
	CODE-SET	Optional	Illegal	Illegal	Illegal
OPEN					
	INPUT	Select one	Select one	Select one	Select one
	REVERSED	Comment	Illegal	Illegal	Illegal
	NO REWIND	Comment	Illegal	Illegal	Illegal
	OUTPUT	Select one	Select one	Select one	Select one
	NO REWIND	Comment	Illegal	Illegal	Illegal
	I-O	Select one	Select one	Select one	Select one
	EXTEND	Optional	Optional	Optional	Illegal
READ					
	NEXT RECORD	Optional	Optional	Optional	Optional
	INTO IDENTIFIER	Optional	Optional	Optional	Optional
	AT END	Optional	Optional	Optional	Optional
	NOT AT END	Optional	Optional	Optional	Optional
	INVALID KEY	Illegal	Optional	Illegal	Optional
	NOT INVALID KEY	Illegal	Optional	Illegal	Optional
	KEY IS	Illegal	Illegal	Optional	Illegal
	INVALID KEY	Illegal	Illegal	Optional	Illegal
	NOT INVALID KEY	Illegal	Illegal	Optional	Illegal

1 This phrase causes the CLOSE statement to be treated as a comment. The file is left open.

Table 5-1. I-O Statements and File Types (continued)

I-O Statement		How the Statement Can Be Used			
Verb	Clause/Phrase	Sequential Organization File	Relative Organization File	Indexed Organization File	Random Access File
REWRITE					
	FROM IDENTIFIER	Optional	Optional	Optional	Optional
	INVALID KEY	Illegal	Optional	Optional	Optional
	NOT INVALID KEY	Illegal	Optional	Optional	Optional
SELECT		Required	Required	Required	Required
	OPTIONAL	Optional	Optional	Optional	Optional
	ASSIGN	Required	Required	Required	Required
	RESERVE	Optional	Optional	Optional	Optional
	ORGANIZATION	Optional	Required	Required	Illegal
	RELATIVE	Illegal	Required	Illegal	
	SEQUENTIAL	Optional	Illegal	Illegal	
	INDEXED	Illegal	Illegal	Required	
	ACCESS MODE	Optional	Optional	Optional	Required
	SEQUENTIAL	Optional	Optional	Optional	Illegal
	RELATIVE KEY	Illegal	Optional	Illegal	
	RANDOM	Illegal	Optional	Optional	Required
	RELATIVE KEY	Illegal	Optional	Optional	Illegal
	ACTUAL KEY	Illegal	Illegal	Illegal	Optional
	DYNAMIC	Illegal	Optional	Optional	Illegal
	RELATIVE KEY		Required	Illegal	
	RECORD KEY	Illegal	Illegal	Required	Illegal
	WITH DUPLICATES			Optional	
	ALTERNATE KEY	Illegal	Illegal	Optional	Illegal
	WITH DUPLICATES			Optional	
	FILE STATUS	Optional	Optional	Optional	Optional
	ACTUAL KEY	Illegal	Illegal	Illegal	Required
SD		Illegal	Illegal	Illegal	Illegal
	RECORD CONTAINS				
	DATA RECORD				
WRITE					
	FROM IDENTIFIER	Optional	Optional	Optional	Optional
	BEFORE ADVANCING	Optional	Illegal	Illegal	Illegal
	AFTER ADVANCING	Optional	Illegal	Illegal	Illegal
	AT END OF PAGE	Optional	Illegal	Illegal	Illegal
	NOT AT END OF PAGE	Optional	Illegal	Illegal	Illegal
	INVALID KEY	Illegal	Optional	Optional	Optional
	NOT INVALID KEY	Illegal	Optional	Optional	Optional

Logical Files

A *logical file* is a data structure that your program declares and accesses. Your program can declare logical files of these four types:

- Sequential organization, including MPE special files.
- Random access.
- Relative organization.
- Indexed organization.

Each file type name reflects the way files of that type are organized and can be accessed. Organization and access method, the major attributes of a file type, determine some of its other attributes.

This section explains the above file types and variable records, which every file type can have.

Table 5-2 summarizes the attributes of the four file types for the purpose of comparison. This section explains each file type in detail. Definitions and explanations of some of the terms in Table 5-2 follow the table.

Table 5-2. Attributes of File Types

Attribute	File Type			
	Sequential Organization	**Random Access**	**Relative Organization**	**Indexed Organization**
Key Type	Does not use keys.	Numeric.	Numeric.	Alphanumeric. Must be written in ascending order if access mode is sequential.
Key Quantity		One	One	1 to 16.
Is Key Unique?		Yes	Yes	No
First Key		Zero	One	Any value.
Open Mode: Input	Yes	Yes	Yes	Yes
Open Mode: Output	Yes	Yes	Yes	Yes
Open Mode: Input-Output	Yes	Yes	Yes	Yes
Open Mode: Extend	Yes	No	Yes	Yes
Sequential Access	Yes	Yes	Yes	Yes
Random Access	No	Yes	Yes	Yes
Dynamic Access	No	No	Yes	Yes
Records can be appended	Yes	Yes	Yes	Yes
Records can be deleted	No	No	Yes	Yes
Records can be inserted	No	Yes	Yes	Yes
Records can be updated	In place.	Yes	Yes	Yes
File portability	Completely portable.	Portable to MPE.	Portable to MPE.	Portable to MPE.
Affects program portability	No	Yes	No	No
Space is allocated for:	Records written.	Records possible (maximum key value plus one, because first key is zero.)	Records possible (maximum key value) and one tag per record.	In Compatibility Mode, two files: one for records written and one for bookkeeping. In Native Mode, one file.
Device on which file can reside:	Any	Disk only.	Disk only.	Disk only.

Below are definitions of the terms in column one of Table 5-2.

Term	Definition
Key	A value within a record that serves to distinguish it from other records.
Input open mode	Allows a program to read a file, but not write it. A file that is open for input access can be used for input, but not output.
Output open mode	Allows a program to write a file, but not read it. A file that is open for output access can be used for output, but not input.
Input-Output open mode	Allows a file to be read and written. A file that is open for input-output access can be used for input and/or output.
Extend open mode	Allows a file to be written in sequential access mode only.
Sequential organization	Allows file records to be read in order from first to last, appended to a file, or written one after another.
Random access	Allows file records to be read or written in any order.
Dynamic access	Allows file to be accessed sequentially or randomly.
Append	To append a record to a file is to add a new record to the end of the file.
Delete	To delete a record from a file is to remove the record from the file.
Insert	To insert a record into a file is to add a new record to the file between two of its existing records (if there is room in the key order).
Update	To update a record is to change its content (without changing its position in the file).
File portability	The degree to which a file is portable; the number of computers on which it can be used, other than the one on which it was created. A completely portable file can be used on any computer. A file that is portable to MPE can be used on any MPE computer (if all its data is USAGE DISPLAY).
Program portability	The degree to which a program is portable; the number of computers on which it can be compiled and/or run, other than the one on which it was originally compiled. File type can affect program portability. A file type that requires nonportable procedures to access it makes programs that use files of that type nonportable.

Sequential Organization Files

A *sequential organization file* is so named because it can only be accessed sequentially. It does not use keys. Because of this simplicity, a sequential organization file:

- Is completely portable (when you are making an ANSI LABELLED TAPE and all data in the records is USAGE DISPLAY).
- Does not limit the portability of the programs that use it.
- Requires space only for records that are actually written to it.
- Can reside on any device.

This section explains the following:

- How to code sequential organization files.
- MPE special files (specialized sequential organization files):
  - Circular files.
  - Message files.
  - Print files.

Table 5-3 lists the access modes for sequential organization files, the open modes associated with them, and the I-O statements that are valid with those open modes.

Table 5-3.
Access Modes, Open Modes, and Valid I-O Statements for Sequential Organization Files

Access Mode	Open Mode	Valid Statements	Explanation
Read-only	`INPUT`	`READ`	You can read the file from beginning to end.
Write-only	`OUTPUT`	`WRITE`	You can write the file from beginning to end. If the file exists, it is overwritten. If it does not exist, it is created.
	`EXTEND`	`WRITE`	You can append records to the file. If the file does not exist, it is created if the SELECT statement specifies the OPTIONAL phrase.
Read-write	`I-O`	`READ, REWRITE; WRITE` if the physical file is a terminal (HP extension)	You can process the file from beginning to end. You can REWRITE a record immediately after a READ (update it in place). You cannot add new records.

A sequential organization file is appropriate for a program that reads or writes a file from beginning to end, without skipping around in it. Examples are transaction files (which are read from beginning to end) and back-up files (which are written from beginning to end).

How to Code Sequential Organization Files

The minimum code your program needs to perform input and output with sequential organization files is:

- In the ENVIRONMENT DIVISION, a SELECT statement with an ASSIGN clause for each file.
- In the FILE SECTION of the DATA DIVISION, an FD entry to match each SELECT statement, with an 01 record for each file.
- In the PROCEDURE DIVISION, procedures to OPEN, READ, WRITE, and CLOSE the files.

Example 1. The following uses a sequential organization file:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FILE-EX1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IFILE            ASSIGN "IFILE".
    SELECT PFILE            ASSIGN "PFILE".
DATA DIVISION.
FILE SECTION.
FD  IFILE.
01  IREC.
    05  NAME               PIC X(30).
    05  SOC-SEC            PIC X(9).
    05  HIRE-DATE.
        10 MO              PIC XX.
        10 DA              PIC XX.
        10 YR              PIC XX.
    05  SALARY             PIC S9(6).
    05                     PIC X(29).
FD  PFILE.
01  PREC.
    05  SOC-SEC            PIC X(9).
    05                     PIC XX.
    05  NAME               PIC X(30).
    05                     PIC XX.
    05  HIRE-DATE.
        10  MO             PIC XX.
        10                 PIC X.
        10  DA             PIC XX.
        10                 PIC X.
        10  YR             PIC XX.
    05                     PIC X(81).
01  HREC.
    05  HSOC-SEC           PIC X(11).
    05  HNAME              PIC X(32).
    05  HHIRE-DATE         PIC X(89).
WORKING-STORAGE SECTION.
01  LNCNT                  PIC S9(4) BINARY VALUE 60.
01  W-DATE.
    05  WYR                PIC XX.
    05                     PIC X(4).
PROCEDURE DIVISION.
P1.
    ACCEPT W-DATE FROM DATE.
    OPEN INPUT IFILE OUTPUT PFILE.
    PERFORM WITH TEST AFTER UNTIL SOC-SEC OF IREC = ALL "9"
        READ IFILE
            AT END MOVE ALL "9" TO SOC-SEC OF IREC
            NOT AT END
```

```
                IF WYR = YR OF IREC THEN
                    ADD 1 TO LNCNT
                    IF LNCNT > 50 PERFORM HEADINGS END-IF
                    MOVE SPACES TO PREC
                    MOVE CORR IREC TO PREC
                    WRITE PREC AFTER ADVANCING 1 LINE
                END-IF
            END-READ
        END-PERFORM
        CLOSE IFILE PFILE
        STOP RUN.
    HEADINGS.
        MOVE "SOC SEC NO" TO HSOC-SEC.
        MOVE "NAME" TO HNAME.
        MOVE "HIRE DATE" TO HHIRE-DATE.
        WRITE PREC AFTER ADVANCING PAGE.
        MOVE 0 TO LNCNT.
```

The following is input to the program:

```
Albert Einstein              343567890010587
James Joyce                  123456789033086
Alice Walker                 987654321020187
Rolando Jiron                333444555121085
```

This program prints the following:

```
SOC SEC NO NAME                          HIRE DATE
343567890  ALBERT EINSTEIN               01 05 87
987654321  ALICE WALKER                  02 01 87
```

The FILE STATUS Clause

The optional FILE STATUS clause specifies a data-item that contains a file status code after any I-O verb (READ, WRITE, OPEN, or CLOSE) is applied to the file. Your program can also contain USE procedures that examine the values of such *data-item*s and perform accordingly. See "File Status Codes" for more information.

The BLOCK CONTAINS Clause

The BLOCK CONTAINS clause is not required. It is better to set the block size outside of the program, when you create the file with the FILE or BUILD command.

The RESERVE Clause

The RESERVE clause specifies the number of file system buffers assigned to a COBOL program at execution time. The default is two buffers, which is optimal for most COBOL programs. A program with extremely heavy I-O and a set of frequently accessed records may perform better with three buffers. Allocating more than three buffers is inefficient use of memory and rarely improves I-O performance.

The CODE-SET Clause

By default, a sequential file contains ASCII data. If your sequential file contains non-ASCII data, you must use the CODE-SET clause to specify its character code convention.

Example. The following program illustrates the CODE-SET clause:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FILE-EX1.
* This program converts an EBCDIC file into an ASCII file.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ALPHABET EBCDIC IS EBCDIC.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IFILE             ASSIGN "IFILE".
    SELECT OFILE             ASSIGN "OFILE".
DATA DIVISION.
FILE SECTION.
FD  IFILE
    CODE-SET IS EBCDIC.
01  IREC                    PIC X(80).
FD  OFILE.
01  OREC                    PIC X(80).
PROCEDURE DIVISION.
P1.
    OPEN INPUT IFILE OUTPUT OFILE.
    PERFORM WITH TEST AFTER  UNTIL IREC = ALL "9"
        READ IFILE
            AT END MOVE ALL "9" TO IREC
            NOT AT END
                MOVE IREC TO OREC
                WRITE OREC
        END-READ
    END-PERFORM
    CLOSE IFILE OFILE.
```

Note A non-ASCII file need not be organized sequentially for a COBOL program to read or write it.

In the preceding examples, the records of each file are of the same format. A file with variable records requires a RECORD clause. See "Variable Records."

Circular Files

A circular file is organized like a sequential organization file, except that it has no "last" record when being written. The record that would be "last" in an ordinary sequential organization file is followed (conceptually) by the record that would be "first."

The diagrams below shows an ordinary sequential organization file with eight records and a circular file with eight records, for comparison.

The following shows an ordinary sequential organization file with eight records.

Record 1	Record 2	Record 3	Record 4	Record 5	Record 6	Record 7	Record 8

The following shows a circular file with eight records (also a sequential organization file):

Record 1	Record 2	Record 3
Record 8		Record 4
Record 7	Record 6	Record 5

The two ways to create a circular file are:

- Use the MPE BUILD command, like this:

  `BUILD` *filename*`;CIR`

- Use the MPE FILE command to cause COBOL to create a circular file, like this:

  `FILE` *filename*`;CIR`

A circular file is appropriate for a history file. A circular file with n records keeps track of the last n transactions, and never fills up.

Example. This example program uses the following:

- A circular file for output.
- A variable record file input. See "Variable Records."
- A SYMBOLIC CHARACTERS clause, an ANSI85 feature. See Chapter 2.

Assuming that the program is in the file named `filex4`, you can use the following sequence of MPE XL commands to run it:

```
:file ifile=$stdin
:file ofile,new;cir;rec=-80,,,ascii;disc=20
:cob85xlg filex4
```

These commands tell the program to do the following:

- Read input from the terminal, unless the program is run from a job stream.
- Write output to a circular file that can hold 20 records.

Note The program does not require a circular output file.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. FILE-EX4.
* Reads input from terminal to variable record file. Writes to circular
* file.  Input consists of commands. Last commands entered can be found
* in circular file. Number of records logged depends on file size.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
    SYMBOLIC CHARACTERS CR IS 14.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT IFILE             ASSIGN "IFILE".
     SELECT OFILE             ASSIGN "OFILE".

 DATA DIVISION.
 FILE SECTION.
 FD  IFILE
     RECORD IS VARYING DEPENDING ON LEN.
 01  IREC.
     05 ICHARS          PIC X OCCURS 0 TO 80 TIMES
                              DEPENDING ON LEN.
 FD  OFILE.
 01  OREC               PIC X(80).

 WORKING-STORAGE SECTION.
 01  LEN                PIC S9(4) BINARY.
 01  ERROR-CODE         PIC S9(4) BINARY.
 01  PARM               PIC S9(4) BINARY.
 PROCEDURE DIVISION.
```

```
P1.
    OPEN INPUT IFILE OUTPUT OFILE.
    PERFORM WITH TEST AFTER UNTIL IREC = "//"
       READ IFILE
          AT END
             MOVE 2 TO LEN
             MOVE "//" TO IREC
          NOT AT END
             IF IREC  "//"
                ADD 1 TO LEN
                MOVE CR TO ICHARS(LEN)
                CALL INTRINSIC "COMMAND"
                     USING IREC ERROR-CODE PARM
                WRITE OREC FROM IREC
             END-IF
       END-READ
    END-PERFORM
    CLOSE IFILE OFILE
    STOP RUN.
```

Message Files

A message file is organized like a sequential organization file that is open for input or output access. Programs use message files to communicate with each other.

The two ways to create a message file are:

- Use the MPE BUILD command, like this:

  `BUILD` *filename*`;MSG`

- Use the MPE FILE command to cause COBOL to create a message file, like this:

  `FILE` *filename*`;MSG`

You must open a message file with the INPUT, OUTPUT, or EXTEND option. Sometimes you must call the FCONTROL intrinsic for full functionality, for example:

- To set a timeout interval.
- To enable an extended wait.

An extended wait allows your program to wait until another program has read or written the message file before it accesses it. For example, your program can wait until another program has written a message to the message file before reading it, or your program can wait until another program has read the message file before writing it.

Example. Suppose that you have a summary job that cannot run until five other jobs have run. The five other jobs are not interdependent; they can run at the same time. The problem is how to ensure that they have run before the summary job runs. One solution is to use a message file. If each of the five jobs logs information into the message file, the summary job can wait until all five jobs have accessed the message file before it executes.

The program for the summary job follows. Notice that it uses FCONTROL to enable extended wait. It calls FCONTROL directly, passing the FD name (IFILE) to the intrinsic as the file number. (Any COBOL program can call any MPE intrinsic in this manner. The access mode of the file is not important.)

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID. FILEX6.
003000 ENVIRONMENT DIVISION.
004000 INPUT-OUTPUT SECTION.
005000 FILE-CONTROL.
006000     SELECT IFILE             ASSIGN "MFILE".
007000 DATA DIVISION.
008000 FILE SECTION.
009000 FD  IFILE.
010000 01  IREC                    PIC X(8).
011000 WORKING-STORAGE SECTION.
012000 01 TRUE-VALUE             PIC 9(4)  BINARY VALUE 1.
013000 01 PROGRAM-COUNT          PIC S9(4) BINARY VALUE 0.
014000 01 DEPENDENCY-TABLE.
015000    05                     PIC X(8) VALUE "FILEX51".
016000    05                     PIC X(8) VALUE "FILEX52".
017000    05                     PIC X(8) VALUE "FILEX53".
018000    05                     PIC X(8) VALUE "FILEX54".
019000    05                     PIC X(8) VALUE "FILEX55".
```

```
020000 01 REDEFINES DEPENDENCY-TABLE.
021000    05 PROGRAM-NAME           PIC X(8) OCCURS 5 TIMES
022000                               ASCENDING KEY IS PROGRAM-NAME
023000                               INDEXED BY I.
024000 PROCEDURE DIVISION.
025000 P1.
026000 WAITING-TO-GO.
027000     OPEN INPUT IFILE.
028000     CALL INTRINSIC "FCONTROL" USING IFILE, 45, TRUE-VALUE.
029000     PERFORM UNTIL PROGRAM-COUNT = 5
030000         READ IFILE
031000            AT END
032000               DISPLAY "AT END on message file should not occur"
033000               GO TO END-OF-PROGRAM
034000         END-READ
035000
036000         SEARCH ALL PROGRAM-NAME
037000            AT END
038000               DISPLAY "filex6: dependency table needs update"
039000               GO TO END-OF-PROGRAM
040000            WHEN PROGRAM-NAME(I) = IREC
041000               ADD 1 TO PROGRAM-COUNT
042000         END-SEARCH
043000
044000     END-PERFORM
045000     CLOSE IFILE.
046000
047000
048000 MAIN-PROGRAM.
049000         DISPLAY "JOBS FILEX51 THRU FILEX55 COMPLETED,"
050000                 " FILEX6 CONTINUING".
051000 END-OF-PROGRAM.
052000         STOP RUN.
053000
```

Each of the five jobs contains code similar to this:

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID. FILEX51.
003000 ENVIRONMENT DIVISION.
004000 INPUT-OUTPUT SECTION.
005000 FILE-CONTROL.
006000     SELECT OFILE            ASSIGN "MFILE".
007000 DATA DIVISION.
008000 FILE SECTION.
009000 FD  OFILE.
010000 01  OREC                   PIC X(8).
011000 PROCEDURE DIVISION.
012000 P1.
013000     OPEN EXTEND OFILE.
014000     MOVE "FILEX51" TO OREC
015000     WRITE OREC
016000     CLOSE OFILE
017000     STOP RUN.
```

This job stream streams the five jobs and the summary job:

```
:job jfilemsg,user.account
:purge mfile
:build mfile;msg;rec=-8,,,ascii;disc=10
:stream ,%
%job jfilex6,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex6
%eoj
%job jfilex51,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex51
%eoj
%job jfilex52,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex52
%eoj
%job jfilex53,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex53
%eoj
%job jfilex54,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex54
%eoj
%job jfilex55,me.myacct/paswd;inpri=7;outclass=pp,3
%file mfile;semi
%run pfilex55
%eoj
:eoj
```

Print Files

A print file is organized like a sequential organization file and has carriage control. The carriage control option cannot be changed after the file is created. See the "WRITE Statement" in the *HP COBOL II/XL Reference Manual* for more information.

Random Access Files

A *random access file* is so named because any record can be accessed at any time by its key. The key corresponds to the record number minus one (for example, key zero is the first record and key four is the fifth record).

The MPE XL operating system does not distinguish between random access and sequential organization files. Therefore, a random access file can be treated like a sequential organization file.

If it is accessed randomly, a random access file has these advantages over a sequential organization file:

- Its records can be accessed in any order.
- New records can be inserted between existing records (as well as appended to the end of the file until the file is full, if there is room in key order).
- Its records can be updated.

A random access file has these disadvantages over a sequential organization file:

- It is not ANSI standard (it is a carryover for COBOL 68 compatibility)
- It must reside on a disk.
- It limits the portability of the programs that access it randomly. (They cannot run on systems that do not have disk storage on which the file can reside, for example.)
- It requires space for every possible record, rather than only for records that are actually written to it.
- Requires fixed length records.

A random access file has these advantages over a relative organization file:

- It is a standard MPE file.
- Non-COBOL programs and routines can read it.
- Access is much faster. WRITE works on a random access file record whether it has been written before or not. On a relative organization file, WRITE only works on new records. Overwriting requires REWRITE.

A random access file has these disadvantages over a relative organization file:

- It is not ANSI standard.
- Its records cannot be deleted.

A random access file can be opened for input, write-only, or input-output access. It is appropriate for a program that must skip around in the file. An example is a record file in which a few random records are updated at a time.

How to Code Random Access Files

The code that your program needs in order to perform input and output with random access files is the following:

1. In the ENVIRONMENT DIVISION, a SELECT statement with an ASSIGN clause for each file. The clauses ACCESS MODE IS RANDOM and ACTUAL KEY are required.
2. In the FILE SECTION of the DATA DIVISION, an FD entry to match each SELECT statement, with an 01 record for each file.
3. In the FILE SECTION or WORKING STORAGE SECTION of the DATA DIVISION, a definition of each data name that you specified in an ACTUAL KEY clause in the ENVIRONMENT DIVISION. Define each data item as an integer large enough to number every record in the file. For maximum efficiency, use PIC S9(9) COMP SYNC.
4. In the PROCEDURE DIVISION, procedures to OPEN, READ, WRITE, REWRITE, and CLOSE the files, and to assign keys (record numbers).

Example. The following program uses random access files.

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID. RANDCRSC.
003000* This program creates a simple random file to show the
004000* blank record created.
005000 ENVIRONMENT DIVISION.
006000 INPUT-OUTPUT SECTION.
007000 FILE-CONTROL.
008000     SELECT RANDFILE ASSIGN TO "RANDFILE"
009000                     ACCESS IS RANDOM
009100                     ACTUAL KEY IS RANDKEY.
010000
011000 DATA DIVISION.
012000 FILE SECTION.
013000 FD RANDFILE.
014000 01 RANDREC.
016000      03   FILLER        PIC  X(10).
016100 WORKING-STORAGE SECTION.
016200 01 RANDKEY    PIC 999.
017000 PROCEDURE DIVISION.
018000 P1.
019000     OPEN OUTPUT RANDFILE.
020000     MOVE 0 TO RANDKEY.
020100     MOVE "000" TO RANDREC.
021000     WRITE RANDREC
022000           INVALID KEY PERFORM ERROR-RTN.
023000     MOVE 6 TO RANDKEY.
023100     MOVE "006" TO RANDREC.
024000     WRITE RANDREC
025000           INVALID KEY PERFORM ERROR-RTN.
026000     MOVE 2 TO RANDKEY.
026100     MOVE "002" TO RANDREC.
027000     WRITE RANDREC
028000           INVALID KEY PERFORM ERROR-RTN.
029000     CLOSE RANDFILE.
030000     STOP RUN.
031000 ERROR-RTN.
032000     DISPLAY "KEY ERROR" RANDREC.
033000     STOP RUN.
```

The preceding program creates a temporary random access file named RANDFILE. The following command displays information about RANDFILE:

```
:LISTFTEMP RANDFILE,2
```

Following is the information displayed:

```
FILENAME  CODE  ------------LOGICAL RECORD------------  ----SPACE----
                  SIZE  TYP        EOF      LIMIT  R/B  SECTORS #X MX

RANDFILE           10B  FA           7      10000   25       16  1  *
```

The following command displays the contents of RANDFILE:

```
:PRINT RANDFILE
```

The above command displays the following:

```
000     Record 1
        Record 2 (blank)
002     Record 3
        Record 4 (blank)
        Record 5 (blank)
        Record 6 (blank)
006     Record 7
```

Assigning Values to Keys

The PROCEDURE DIVISION must include a procedure that assigns a key to the data item specified by the ACTUAL KEY clause for each record in the file. When you write this key-assigning procedure, keep in mind:

- The first record must receive the number zero.
- It is an error to assign a record number that is greater than the number of records the file can hold. (This number is determined when the file is created with the FILE or BUILD command.)
- A random access file is allocated space for every possible key, so avoid assigning keys in a way that creates a lot of unused space. For example, if you assign keys 1000-9999, dummy records 0-999 will be allocated. If you never intend to use records 0-999, they waste space.
- If the keys do not naturally fall within record number zero and the last record in the file, you must devise an algorithm to compute the record key — a hashing algorithm, for example.

A *hashing algorithm* is any algorithm that maps larger numbers to smaller numbers. The following are some characteristics of hashing algorithms:

- Hashing algorithms often use modular arithmetic.
- They add the digits of a larger number together to produce a smaller number.
- They map nonnumeric data to numbers and then map those numbers to smaller numbers.

For more information on hashing algorithms, refer to a book on computer algorithms. For an example of a hashing algorithm, see the example in "Relative Organization Files."

Accessing Random Access Files Sequentially

The MPE XL operating system does not distinguish between random access and sequential organization files. The distinction is made by the HP COBOL II/XL compiler, which generates different code for random access files. This means that a file can be created as a sequential organization file by one program and treated as a random access file by another program, or vice versa.

When a random access file is treated as a sequential organization file, its records are accessed sequentially, beginning with record zero. The file system cannot distinguish between dummy records and real records, however. If the random access file is an ASCII file, its dummy records contain spaces; if it is a binary file, its dummy records contain binary zeros. Dummy records are treated as data records.

Relative Organization Files

A *relative organization file* is so named because the relative order of its records is constant. It has MPE XL file type RIO. It differs from a random access file only in the following ways:

- Its first key is one, not zero.
- Its records can be deleted.
- It is portable to MPE computers (not completely portable).
- It does not limit the portability of the programs that use it, because it is ANSI Standard.
- In addition to space for all possible records, it requires space for one tag per record (the tag indicates whether the associated record has been deleted).
- It is not interchangeable with a sequential organization file. That is, it cannot be created as a sequential file by one program and accessed as a relative organization file by another, or vice versa. The reason is that a relative organization file has an extra bit allocated for each record. Each bit indicates whether its record has been deleted.
- It uses disk space less efficiently than a random access file.
- I-O is less efficient than it is on a random access file.
- Simulates variable length records (with fixed length records) transparently.

How to Code Relative Organization Files

The code that your program needs to perform input and output with relative organization files depends on how you want to access the file. You can access a relative organization file sequentially, randomly, or dynamically.

Sequential Access. Sequential access is the default. You may specify ACCESS IS SEQUENTIAL in the SELECT statement or omit it. Sequential access allows you to access records in order. Use sequential access when you want to move forward in the file, but never backward. For example, a batch payroll program that processes every employee record would use sequential access.

Random Access. Random access requires that you specify ACCESS IS RANDOM in the SELECT statement. Random access allows you to access records by key. Use random access when you want to access records in any order. For example, a personnel application program that retrieves employee records by employee number would use random access.

Dynamic Access. Dynamic access requires that you specify ACCESS IS DYNAMIC in the SELECT statement. Dynamic access allows you to skip to any record, before or after the current record, and then access the file sequentially from there. Use dynamic access when you want to access records in order after a certain record. For example, a program that reads every record written after a certain date, where the date is the key, would use dynamic access.

Relative Organization Files

Table 5-4 lists the access modes for relative organization files, the open modes associated with them, the I-O statements that are valid with those open modes, and the phrases that are valid and invalid with those I-O statements.

Table 5-4.
Acess Modes, Open Modes, and Valid I-O Statements for Relative Organization Files

Access Mode	Open Mode	Valid Statements	Valid Phrases	Invalid Phrases
Sequential	INPUT	READ	NEXT, INTO, AT END, NOT AT END, END-READ	INVALID KEY, NOT INVALID KEY
		START	All phrases.	None.
	OUTPUT	WRITE	All phrases.	None.
	I-O	READ	NEXT, INTO, AT END, NOT AT END, END-READ	INVALID KEY, NOT INVALID KEY
		REWRITE	FROM, END-REWRITE	INVALID KEY, NOT INVALID KEY
		START	All phrases.	None.
		DELETE	END-DELETE	INVALID KEY, NOT INVALID KEY
	EXTEND	WRITE	All phrases.	None.

Table 5-4.
Acess Modes, Open Modes, and Valid I-O Statements for Relative Organization Files (continued)

Access Mode	Open Mode	Valid Statements	Valid Phrases	Invalid Phrases
Random	INPUT	READ	INTO, INVALID KEY, NOT INVALID KEY, END-READ	AT END, NOT AT END
	OUTPUT	WRITE	All phrases.	None.
	I-O	READ	INTO, INVALID KEY, NOT INVALID KEY, END-READ	AT END, NOT AT END
		REWRITE, DELETE, WRITE	All phrases.	None.
Dynamic	INPUT	READ	All phrases. (NEXT is required to show that records are to be accessed sequentially.)	None.
		START	All phrases.	None.
	OUTPUT	WRITE	All phrases.	None.
	I-O	READ, REWRITE, START, DELETE, WRITE	All phrases.	None.

As with a random access file, a relative organization file may require that the program have a hashing algorithm.

Example. This program writes records to a relative file, using a simple hashing algorithm to map Social Security numbers to three-digit numbers. The hashing algorithm uses the last three digits of the Social Security number as the key. If there is a duplicate key, the hashing algorithm adds one to the key until it is unique.

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID. HASHSC.
003000* This program writes records to a relative file using
004000* a hashing scheme.
005000
006000 ENVIRONMENT DIVISION.
007000 INPUT-OUTPUT SECTION.
008000 FILE-CONTROL.
009000     SELECT RELFILE  ASSIGN TO "RELFILE"
010000                     ORGANIZATION IS RELATIVE
011000                     ACCESS IS RANDOM
012000                     RELATIVE KEY IS W-KEY.
013000
014000 DATA DIVISION.
015000 FILE SECTION.
016000 FD RELFILE.
017000 01 REL-REC.
018000     03  REL-SS-NO      PIC  X(9).
019000 WORKING-STORAGE SECTION.
020000 01  IN-SS-NO.
021000     88  NO-MORE-SS-NUMBERS  VALUE ALL "9".
022000     05  FILLER         PIC X(6).
023000     05  IN-KEY         PIC 999.
024000 01  WRITE-SWITCH       PIC XXX.
025000     88  SUCCESS        VALUE "YES".
026000     88  RESET-SWITCH   VALUE "NO".
027000 01  W-KEY              PIC 999.
028000
```

```
029000 PROCEDURE DIVISION.
030000 000-MAIN-PROG.
031000     OPEN OUTPUT RELFILE.
032000     PERFORM 100-GET-SS
033000     PERFORM WITH TEST AFTER UNTIL NO-MORE-SS-NUMBERS
034000        PERFORM 200-WRITE-TO-RELFILE UNTIL SUCCESS
035000        SET RESET-SWITCH TO TRUE
036000        PERFORM 100-GET-SS
037000     END-PERFORM
038000     CLOSE RELFILE.
039000     STOP RUN.
040000
041000 100-GET-SS.
042000     ACCEPT IN-SS-NO.
043000
044000 200-WRITE-TO-RELFILE.
045000     MOVE IN-KEY TO W-KEY
046000     MOVE IN-SS-NO TO REL-SS-NO
047000     PERFORM UNTIL SUCCESS
048000         WRITE REL-REC
049000             INVALID KEY     ADD 1 TO W-KEY
050000             NOT INVALID KEY SET SUCCESS TO TRUE
051000         END-WRITE
052000     END-PERFORM.
053000
054000 999-LAST-PARA.
055000     DISPLAY "ERROR -- FELL OFF THE END OF THE PROGRAM".
```

This program creates a file that contains the following data:

```
Social security number 333444001 is record number 001
Social security number 123423007 is record number 007
Social security number 111222008 is record number 008
Social security number 123456009 is record number 009
Social security number 222333007 is record number 010
Social security number 444555010 is record number 011
```

Indexed Organization Files

An indexed file is so named because each record has an index, which is an alphanumeric key. An indexed file is organized as an MPE XL KSAM file is organized. You can access both Native Mode and Compatibility Mode KSAM files from HP COBOL II/XL programs. Refer to *Using KSAM/XL* and the *KSAM/3000 Reference Manual* for details.

An indexed file differs from a relative organization file only in the following ways:

- The primary keys of an indexed file can be alphanumeric and must be written in ascending order (according to ASCII value) if access mode is sequential and open mode is OUTPUT or EXTEND. The primary keys must be USAGE DISPLAY.
- An indexed file can specify one to 16 keys.
- The indexed file's first key can be any value.
- Its keys need not be unique. Two records can have the same value for one key only if they have different values for another key. It is recommended that primary keys be unique.
- Two files are required for support of Compatibility Mode KSAM files, one for data and one for the index. The data file has space only for records written, not for every possible record. For Native Mode KSAM files, only one file is required.

An indexed file is appropriate when alphanumeric keys are appropriate or keys are not unique. An example is an employee file where the employees' surnames are the keys. The surnames are alphanumeric and two or more employees can have the same surname.

An indexed file is also appropriate for a sparse file. If you write a record with the index "A" followed by a record with the index "Z" space is not allocated for the records that could be inserted between them.

How to Code Indexed Organization Files

Indexed files can be accessed sequentially, randomly, or dynamically. The code that your program needs to access indexed files sequentially is the following:

1. In the ENVIRONMENT DIVISION, a SELECT statement with an ASSIGN clause for each file. The clauses RECORD KEY and ORGANIZATION IS INDEXED are required.
2. In the FILE SECTION of the DATA DIVISION, an FD entry to match each SELECT statement, with an 01 record for each file. In the FD entry, none of the clauses are required. The 01 record must contain the data item specified in the RECORD KEY clause in the ENVIRONMENT DIVISION.
3. In the PROCEDURE DIVISION, procedures to OPEN, READ, WRITE, REWRITE, DELETE, and CLOSE the files, and to assign keys, or record numbers.

To access indexed files randomly or dynamically, add the clause ACCESS MODE IS RANDOM or ACCESS MODE IS DYNAMIC to the SELECT statement.

Example. The following program uses an indexed organization file:

```
001000 IDENTIFICATION DIVISION.
002000 PROGRAM-ID. INDEXSC.
003000* This program reads an indexed file dynamically.  It does a
004000* random read to get to the specific ALUM record desired, then
005000* it reads sequentially forward through the records of that alumnus
006000* to total all the gifts made by that alumnus.
007000 ENVIRONMENT DIVISION.
008000 INPUT-OUTPUT SECTION.
009000 FILE-CONTROL.
010000     SELECT ALUMFILE ASSIGN TO "ALUMFILE"
011000            ORGANIZATION IS INDEXED
012000            RECORD KEY IS ID-NUMBER WITH DUPLICATES
013000            ACCESS MODE IS DYNAMIC.
014000     SELECT PRINT-FILE ASSIGN TO "PFILE".
015000 DATA DIVISION.
016000 FILE SECTION.
017000 FD ALUMFILE.
018000 01 ALUM-REC.
019000      03 ID-NUMBER       PIC X(9).
020000         88 NO-MORE-ALUMS VALUE "//".
021000      03 NAME            PIC X(20).
022000      03 GIFT            PIC S9(6)V99.
023000      03 FILLER          PIC X(43).
024000 FD PRINT-FILE.
025000 01 PRINT-REC.
026000      03 P-ID-NUMBER          PIC X(9).
027000      03                     PIC X.
028000      03 P-NAME               PIC X(20).
029000      03                     PIC X.
030000      03 P-TOTAL-GIFTS        PIC $$,$$$,$$$.$$.
031000 WORKING-STORAGE SECTION.
032000 01 EOF-SW               PIC X   VALUE "N".
033000    88 ALUM-EOF          VALUE "Y".
034000 01 W-TOTAL              PIC S9(7)V99 VALUE 0.
035000 01 HOLD-ALUM.
036000      03 H-ID-NUMBER     PIC X(9).
037000      03 H-NAME          PIC X(20).
038000      03 H-GIFT          PIC S9(6)V99.
039000      03 FILLER          PIC X(43).
```

```
040000 PROCEDURE DIVISION.
041000 000-MAIN-PROG.
042000     OPEN INPUT ALUMFILE OUTPUT PRINT-FILE.
043000     PERFORM UNTIL NO-MORE-ALUMS
044000         PERFORM 100-WHICH-ALUM
045000         IF NOT NO-MORE-ALUMS
046000             PERFORM 150-TOTAL-THE-ALUMS-GIFTS
047000             END-IF
048000     END-PERFORM.
049000     STOP RUN.
050000
051000 100-WHICH-ALUM.
052000     ACCEPT ID-NUMBER.
053000
054000 150-TOTAL-THE-ALUMS-GIFTS.
055000     READ ALUMFILE INTO HOLD-ALUM
056000         KEY IS ID-NUMBER
057000         INVALID KEY PERFORM 200-NO-SUCH-ALUM
058000         NOT INVALID KEY
059000             MOVE ZERO   TO W-TOTAL
060000             PERFORM UNTIL H-ID-NUMBER NOT = ID-NUMBER
061000                 ADD GIFT TO W-TOTAL
062000                 PERFORM 175-SEQUENTIAL-READ
063000             END-PERFORM
064000         END-READ.
065000     PERFORM 300-PRINT-OUT-ALUM.
066000
067000 175-SEQUENTIAL-READ.
068000     READ ALUMFILE NEXT RECORD
069000         AT END MOVE ALL "X" TO ID-NUMBER
070000         END-READ.
071000 200-NO-SUCH-ALUM.
072000     MOVE "                  THIS ID NUMBER NOT IN ALUMNI FILE"
073000           TO PRINT-REC
074000     MOVE ID-NUMBER TO P-ID-NUMBER
075000     WRITE PRINT-REC.
076000
077000 300-PRINT-OUT-ALUM.
078000     MOVE H-ID-NUMBER TO P-ID-NUMBER
079000     MOVE H-NAME TO P-NAME
080000     MOVE W-TOTAL TO P-TOTAL-GIFTS
081000     WRITE PRINT-REC.
```

Creating Indexed Files

If your program is to use an indexed file that does not exist, HP COBOL II/XL (Native Mode) by default creates a temporary Native Mode KSAM file—one file. If an indexed file has variable length records, HP COBOL II/XL creates a temporary Compatibility Mode KSAM file.

HP COBOL II/V (Compatibility Mode) by default creates a temporary Compatibility Mode KSAM file. Compatibility Mode KSAM files consist of two temporary files: a data file and an index file. The data file has the name that the ASSIGN clause specifies. The index file has the same name, with "K" appended to it. If the data file name has eight characters, the index file name has the same first seven characters and "K" as the eighth character. In the SELECT statement, include the clause ACCESS MODE IS SEQUENTIAL.

You can force an HP COBOL II/V program to create a Native Mode KSAM file by using a FILE equation with the ";KSAMXL" option.

Alternatively, you can create an indexed file before you execute your program. To create a Compatibility Mode KSAM file, use the >BUILD command of the KSAM utility KSAMUTIL. To create a Native Mode KSAM file, use the MPE XL :BUILD command with the ";KSAMXL" option. Both HP COBOL II/V and HP COBOL II/XL programs can access existing Native Mode KSAM files and existing Compatibility Mode KSAM files. For details, refer to *Using KSAM/XL* and the *KSAM/3000 Reference Manual*.

Example. If the following program is compiled with HP COBOL II/XL, the run-time library creates the Native Mode KSAM file *NFILE*. If the program is compiled with HP COBOL II/V, the run-time library creates a Compatibility Mode KSAM file consising of two temporary files, *NFILE* and *NFILEK*. The program copies the records from the sequential file to the indexed file, using the first six characters in each record as its key.

```
001100 IDENTIFICATION DIVISION.
001200 PROGRAM-ID. TFILE.
001500 ENVIRONMENT DIVISION.
001900 INPUT-OUTPUT SECTION.
002000 FILE-CONTROL.
002100     SELECT NEWFILE
002110            ASSIGN TO "NFILE"
002120            ORGANIZATION IS INDEXED
002130            ACCESS MODE IS SEQUENTIAL
002140            RECORD KEY IS NKEY WITH DUPLICATES.
002200     SELECT TFILE ASSIGN TO OFILE.
002300 DATA DIVISION.
002400 FILE SECTION.
002500 FD NEWFILE.
002600 01 NREC.
002610    05 NKEY PIC X(6).
002620    05   PIC X(74).
002700 FD TFILE.
002800 01 TREC.
002810    05 TKEY PIC X(6).
002820    05   PIC X(74).
003200 PROCEDURE DIVISION.
003300 P1.
003400      OPEN INPUT TFILE OUTPUT NEWFILE
```

```
003410      READ TFILE AT END MOVE ALL "9" TO TREC END-READ
003500      PERFORM UNTIL TREC = ALL "9"
003600          WRITE NREC    FROM TREC
003610               INVALID KEY DISPLAY "ERROR" TREC
003620               END-WRITE
003630          READ TFILE AT END MOVE ALL "9" TO TREC END-READ
003700      END-PERFORM
003710      CLOSE TFILE NEWFILE
003780      STOP RUN.
```

Sequential Access of Indexed Files

For an indexed file, sequential access is in ascending key order. Attempting to write a record out of order causes an invalid key error at run time (file status code 21).

A sequential rewrite to an indexed file updates the record that was read immediately before the REWRITE statement executed. If the REWRITE statement was not immediately preceded by a READ statement, a logic error occurs.

The following table gives the modes in which you must open an indexed file to perform the operations READ, WRITE, REWRITE, and DELETE. The KEY IS phrase is not required.

Table 5-5. Modes to Open Indexed Files for Sequential Access

Operation	Modes in Which to Open File
READ	INPUT or I-O
WRITE	OUTPUT, EXTEND, or I-O
REWRITE	I-O
DELETE	I-O

Random and Dynamic Access

For an indexed file, random and dynamic access use the primary key to determine which record to WRITE, REWRITE, or DELETE.

Generic Keys

A generic key is a partial key, the first *n* digits of a key. It is specified in the START statement.

Example. In the following example, SOURCE-NO is a generic key. The START statement finds the first record that contains the generic key.

```
SELECT I-FILE ASSIGN TO "INVFILE"
   ORGANIZATION IS INDEXED
   RECORD KEY IS PART-NO.
01 IREC.
   05 PART-NO.
      10 SOURCE-NO         PIC XXXX.
      10 ITEM-NO           PIC XXXX.
   05 FILLER               PIC X(72).

START I-FILE KEY = SOURCE-NO
   INVALID KEY DISPLAY "RECORD NOT FOUND"
   END-START.
```

Duplicate Keys

An indexed file can have duplicate primary keys if the WITH DUPLICATES phrase is specified. It can be specified for both the RECORD KEY and ALTERNATE KEY clauses. (Allowing the WITH DUPLICATES phrase with the RECORD KEY clause is an HP extension.)

Duplicate primary keys severely limit the functions that can be performed on the file. In random or dynamic access, DELETE and REWRITE statements apply only to the first record that has the specified primary key.

Variable Length Records

Variable length records are allowed in every logical file organization.

For random access files and relative organization files, HP COBOL II simulates variable length records by using fixed length records. HP COBOL II builds the file with a record size two bytes longer than the largest logical record, rounded up to a two-byte boundary, defined by the program. No space is saved. The file must be created by an HP COBOL II program with the same file characteristics as the program that will access the file. When creating a random access file with variable length records, use a file equation like the following to force the creation of fixed length records:

```
:FILE X;REC=,,F
```

Variable length record files created outside of HP COBOL II can be accessed with ORGANIZATION SEQUENTIAL.

HP COBOL II directly supports variable length records in indexed and sequential organization files.

Specify variable length records with the RECORD IS VARYING clause of the FD level indicator. The following lists methods of specifying variable length records that are not recommended:

Method Not Recommended	**Reason**
RECORDING MODE IS V	The RECORDING MODE clause is not ANSI standard.
RECORD CONTAINS *integer-4* TO *integer-5* CHARACTERS	The RECORD CONTAINS clause does not necessarily create a variable record file. It only causes the compiler to check that the record size is between the values *integer-4* and *integer-5*.

When reading variable length records, do one of the following:

- In the FILE SECTION, define the record as OCCURS DEPENDING ON, using the same data item as you use in the DEPENDING ON phrase of the RECORD IS VARYING and OCCURS clauses.
- Use a READ ... INTO statement to blank out the unfilled part of the record in the WORKING-STORAGE SECTION.
- Before each READ statement executes, blank out the record associated with the FD level indicator.

Any one of the above ensures that when you overwrite a larger record value with a smaller one, the record value does not retain "extra" information from the larger record.

Variable length records are appropriate for any of the following:

- Saving file space on disk. In memory, the maximum space is allocated, but on disk, the actual size is allocated.
- Reading tapes of unknown format or MPE records of undefined-length. If this is the case, use the following file equation:

  `:FILE` *logical_filename*`;DEV=TAPE;REC=,,U`

- Terminal I-O, because terminal records are always of undefined length. This enables you to know how many characters were entered from the terminal.

  Every time you read from a variable length or undefined-length file, the READ statement assigns the number of characters read to the data name in the DEPENDING ON phrase of the RECORD IS VARYING clause.

  The terminal is the physical file that is associated with the logical file. One way to make this association is with the following file equation:

  `:FILE` *logical_file_name*`;DEV=TERM`

- The number of characters written is the exact number of characters specified by the data name in the DEPENDING ON phrase.

  An alternative to a variable length record (RECORD IS VARYING) that also ensures that the exact number of characters is written is to define the record in the FILE SECTION with an OCCURS DEPENDING ON clause.

Example

The following program illustrates variable length records. It opens the same file with and without the DEPENDING ON phrase. Note that in EXAMPLE 2, IREC2 is not overwritten by a READ statement, but READ INTO does overwrite the working storage record.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBVAR.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT IFILE ASSIGN TO "IFILE".
SELECT IFILE2 ASSIGN TO "IFILE".
DATA DIVISION.
FILE SECTION.
FD IFILE
   RECORD IS VARYING FROM 10 TO 50 DEPENDING ON LEN.
   01 IREC.
   05 FILLER     PIC X OCCURS 10 TO 50 TIMES DEPENDING ON LEN.
FD IFILE2
   RECORD IS VARYING FROM 10 TO 50.
01 IREC2         PIC X(50).
WORKING-STORAGE SECTION.
01 LEN           PIC S9(4) BINARY.
01 WREC          PIC X(50).
PROCEDURE DIVISION.
P1.
    DISPLAY "EXAMPLE 1 OCCURS DEPENDING ON REC"
    OPEN INPUT IFILE
    PERFORM UNTIL LEN = -1
        READ IFILE
             AT END MOVE -1 TO LEN
             NOT AT END
                 DISPLAY IREC
                 DISPLAY LEN
        END-READ
    END-PERFORM
    CLOSE IFILE
    DISPLAY SPACE

    DISPLAY "EXAMPLE 2 FIXED REC"
    OPEN INPUT IFILE2
    MOVE ALL "X" TO IREC2
    READ IFILE2 AT END MOVE -1 TO LEN
    DISPLAY IREC2
    DISPLAY SPACE
```

```
DISPLAY "EXAMPLE 3 READ INTO WREC"
MOVE ALL "X" TO IREC2 WREC
READ IFILE2 INTO WREC AT END MOVE -1 TO LEN
DISPLAY IREC2
DISPLAY WREC
CLOSE IFILE2.
```

Assume that IFILE contains the following data:

```
1234567890
123456789*123456789*
123456789*123456789*123456789*
```

The preceding program displays the following:

```
EXAMPLE 1 OCCURS DEPENDING ON REC
1234567890
+00010
123456789*123456789*
+00020
123456789*123456789*123456789*
+00030

EXAMPLE 2 FIXED REC
1234567890XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

EXAMPLE 3 READ INTO WREC
123456789*123456789*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
123456789*123456789*
```

Physical Files

A *physical file* exists outside your program, on a device such as a tape or disk. In contrast, a *logical file* is a data structure that your program declares. When you associate a physical file with a logical file and execute your program, everything that your program does to the logical file happens to the physical file.

This section explains the following:

- The ASSIGN clause, which associates a logical file with a physical file.
- Temporary physical files, which exist only while your program is executing.
- The MPE BUILD command, which creates a physical file.
- The MPE FILE command, which associates a physical file with a logical file and can specify new attributes for the logical file.
- How to dynamically associate a logical file with a physical file at run time with the USING clause.
- How to access multiple physical files on a labelled tape without rewinding the tape.

Figure 5-1 shows how the COBOL runtime library determines which physical file to open when it executes an OPEN statement.

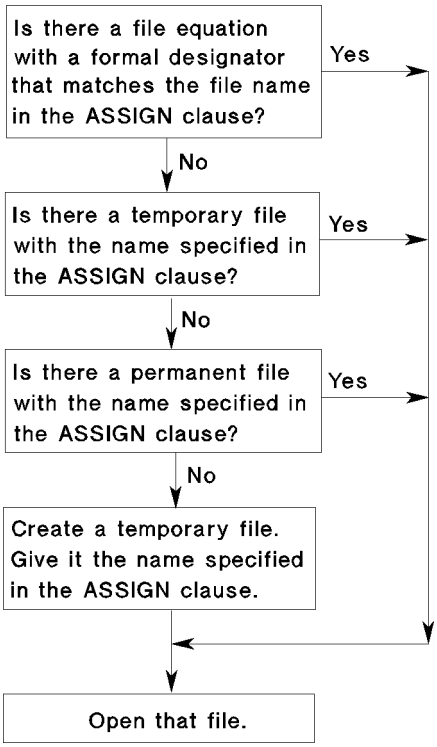


Figure 5-1. Algorithm for Determining Which File to Open

ASSIGN Clause

The ASSIGN clause associates a logical file with a physical file. It is required for files of all types.

The OPEN statement looks for a physical file named in the ASSIGN clause to associate with the logical file. It looks for these physical files in this order:

1. A file equation first.
2. A temporary physical file second.
3. A permanent physical file last.

Note If present, the contents of the item in the USING phrase is used instead of the literal in the ASSIGN clause. See "Dynamic Files (USING phrase)" below for details.

Temporary Physical Files

If you need a physical file only while your program executes, you can use a temporary physical file. You can obtain a temporary physical file in either of these two ways:

- Let COBOL create the temporary physical file automatically when you execute your program. The attributes of the temporary physical file are those that you specify with the FD and SELECT statements. (This applies only to old OUTPUT, EXTEND, and I-O files.)
- Use the MPE command BUILD or FILE, specifying TEMP.

BUILD Command

The MPE BUILD command creates a physical file when it is executed (as opposed to specifying the attributes of a file that will be created when a program opens it). Your program can use such a physical file by opening it with the OPEN statement.

For more information on the BUILD command, refer to the *HP COBOL II/XL Reference Manual* and the *MPE XL Commands Reference Manual*.

FILE Command

The MPE FILE command, also called a file equation, performs one or both of the following:

- Associates a physical file in the MPE environment with the logical file defined by your program.
- Specifies attributes for the logical file, overriding the name and other attributes that the program specifies in the OPEN statement.

The FILE command can override the following attributes:

- File name, including optional node name.
- File size, including number of extents (new files only).
- File type (CIR, MSG, or STD) (new files only).
- Block size (new files only).
- Whether the file is ASCII or binary (new files only).
- Whether the file has carriage control (new files only).
- Whether access is exclusive or shared.
- Number of input/output buffers to be assigned to the file.
- File disposition, which is what happens to the file after it is closed.
- Device.
- Output priority.
- Number of copies to be printed.
- Whether the magnetic tape that contains the file is labelled.

Note The HP COBOL II/XL compiler requires a closer match between physical file attributes and program-specified attributes when invoked through its ANSI85 entry point than it does when invoked through its ANSI74 entry point. Attempting to override a physical file attribute with a FILE equation causes permanent error 39. See Table 5-6.

If you want to use one or more FILE commands, execute them before you execute your program. See Chapter 6.

For more information on the FILE command, refer to the *MPE XL Commands Reference Manual*.

Figure 5-2 shows the algorithm that the run-time library uses to determine file attributes when it opens a file. For the default attributes of the :FILE command, see this command in the *MPE XL Commands Reference Manual*.

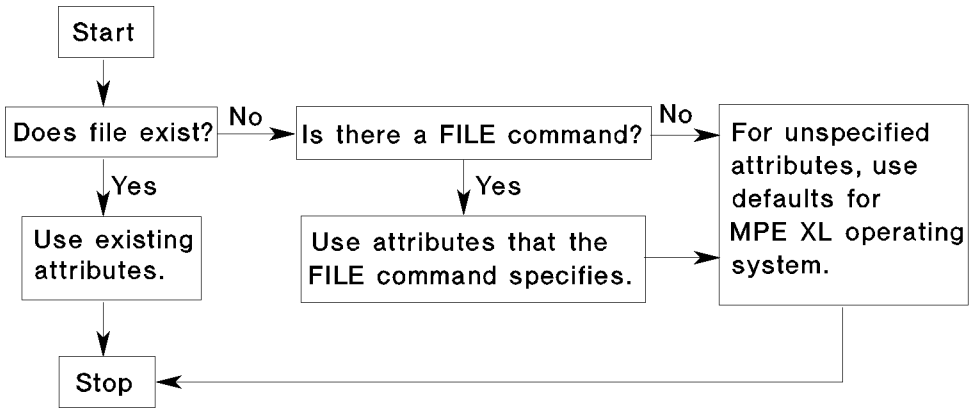


Figure 5-2. Algorithm for Determining File Attributes

Some file attributes are fixed. That is, they are established when the file is created and cannot be changed. The fixed file attributes are:

- Organization.
- Alternate record key.
- Primary record key.
- Code set.
- Minimum and maximum record sizes.
- Record type, fixed or variable.
- Collating sequence of keys in an indexed file.
- Blocking factor.

The enforcement of fixed file attributes in ANSI COBOL 1974 were less stringent. If you have a fixed file attribute conflict, do one of the following:

- Change your program, matching the attributes of the logical file to the fixed attributes of the physical file.
- Recompile your program with the STAT74 control option. The compiler will use the less stringent checking of ANSI COBOL 1974, but you can still use its ANSI85 entry point.
- Recompile your program with the ANSI74 entry point.

Dynamic Files (USING phrase)

You can assign a logical file to a physical file dynamically, at run time. Normally, you assign the physical file statically in the ASSIGN clause by naming the file in the TO phrase. With the USING phrase of the ASSIGN clause, instead of specifying the name of the physical file, you specify a data item to contain the name of the physical file. You can then change the value of the data item at run time to open different physical files. However, for each logical file, only one physical file can be open at a time.

For more information on the ASSIGN clause, see the *HP COBOL II/XL Reference Manual*.

Example

The following program uses dynamic file assignment. The program reads a file name from the terminal, opens the file, and displays its contents one screen at a time. It then reads another file name from the terminal and displays that file. This program can display any number of files with like fixed file attributes.

```
001000 IDENTIFICATION DIVISION.
001100 PROGRAM-ID.  DYNFILE.
001200 ENVIRONMENT DIVISION.
001300 INPUT-OUTPUT SECTION.
001400 FILE-CONTROL.
001500 SELECT INFILE ASSIGN USING FILE-NAME
001600               FILE STATUS IS INFILE-STAT.
001700 DATA DIVISION.
001800 FILE SECTION.
001900 FD INFILE.
002000 01  IN-RECORD                PIC X(80).
002100 WORKING-STORAGE SECTION.
002200 01  FILE-NAME       PIC X(9) VALUE SPACES.
002300     88 NO-MORE-FILES         VALUE "//".
002400 01  SCREEN-SIZE     PIC 99 VALUE ZERO.
002500     88 SCREEN-FULL          VALUE 22.
002600 01  RETURN-KEY      PIC X.
002700 01  INFILE-STAT     PIC XX.
002800     88 INFILE-EOF         VALUE "10".
002900     88 INFILE-NOT-THERE   VALUE "35".
```

```
003000 PROCEDURE DIVISION.
003100 FIRST-PARA.
003200    PERFORM UNTIL NO-MORE-FILES
003300       MOVE SPACES TO FILE-NAME, INFILE-STAT
003400       DISPLAY "Enter the name of the file to list."
003500       DISPLAY "Enter // if no more files: "
003600           WITH NO ADVANCING
003700       ACCEPT FILE-NAME
003800       IF NO-MORE-FILES
003900          STOP RUN
004000       ELSE
004100          PERFORM GET-FILE
004200       END-IF
004300    END-PERFORM.
004400
004500 GET-FILE.
004600    OPEN INPUT INFILE
004700    EVALUATE INFILE-STAT
004800        WHEN "35" DISPLAY "Could not find the file."
004900        WHEN "00" PERFORM DISPLAY-FILE
005000        WHEN OTHER
005100               DISPLAY "An error occurred while opening the file."
005200    END-EVALUATE.
005300
005400 DISPLAY-FILE.
005500    PERFORM UNTIL INFILE-EOF
005600        MOVE ZERO TO SCREEN-SIZE
005700        PERFORM UNTIL SCREEN-FULL
005800           READ INFILE
005900              AT END SET SCREEN-FULL TO TRUE
006000              NOT AT END
006100                 DISPLAY IN-RECORD WITH NO ADVANCING
006200                 ADD 1 TO SCREEN-SIZE
006300           END-READ
006400        END-PERFORM
006500        DISPLAY "Press Return: " WITH NO ADVANCING
006600        ACCEPT RETURN-KEY
006700    END-PERFORM
006800    CLOSE INFILE.
```

Below is sample output from the above program.

```
Enter the name of the file to list.          The program asks for a file name.
Enter // if no more files: FILE1             File name is FILE1.
Here is line 1 of FILE1.                     Contents of FILE1.
Here is line 2 of FILE1.                     Contents of FILE1.
Press Return:                                End of FILE1.
Enter the name of the file to list.          The program asks for another file name.
Enter // if no more files: FILEA             File name is FILEA.
Here is line 1 of FILEA.                     Contents of FILEA.
Here is line 2 of FILEA.                     Contents of FILEA.
Here is line 3 of FILEA.                     Contents of FILEA.
Here is line 4 of FILEA.                     Contents of FILEA.
Press Return:                                End of FILEA.
Enter the name of the file to list.          The program asks for another file name.
Enter // if no more files: FILE2             File name is FILE2.
Could not find the file.                     FILE2 does not exist.
Enter the name of the file to list.          The program asks for another file name.
Enter // if no more files: FILE3             File name is FILE3.
An error occurred while opening the file.    FILE3 has 72-character records.
Enter the name of the file to list.          The program asks for another file name.
Enter // if no more files: //                Enter // to end the program.
:                                            Return to the MPE XL prompt.
```

Multiple Files on a Labelled Tape

Two obsolete features of the ANSI 1985 Standard allow you to sequentially access multiple physical files on a labelled tape, without rewinding the tape. They are the MULTIPLE FILE TAPE clause in the I-O CONTROL paragraph and the VALUE OF clause in the file descriptor. Either one, used in conjunction with the CLOSE WITH NO REWIND statement, allows such access.

Because the MULTIPLE FILE TAPE and VALUE OF clauses are obsolete, a file equation of the following form is recommended instead:

```
FILE filename; LABEL=volume_id, type, expiration_date, NEXT
```

As with the obsolete clauses, the above file equation may be used in conjunction with the CLOSE WITH NO REWIND statement.

Overwriting Files

When you overwrite an existing file, you cannot make it larger unless it is on tape. If it is not on tape, space is limited to the original allocation.

Updating Files

You can update a sequential file with the REWRITE statement if the file resides on a disk. If it resides on tape, you cannot update it.

Appending to Files

To append records to a file, you must include the EXTEND phrase in the OPEN statement. You can add new records until the space originally allocated to the file is filled.

Example

The following program appends records to a file.

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. FILE-EX3.
  * This program appends records to a file.
  * File ifile will be concatenated to the end of ofile.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
        SELECT IFILE             ASSIGN "IFILE".
        SELECT OFILE             ASSIGN "OFILE".
    DATA DIVISION.
    FILE SECTION.
    FD  IFILE.
    01  IREC                    PIC X(80).
    FD  OFILE.
    01  OREC                    PIC X(80).
    PROCEDURE DIVISION.
    P1.
       OPEN INPUT IFILE EXTEND OFILE.
       PERFORM WITH TEST AFTER UNTIL IREC = ALL "9"
           READ IFILE
               AT END MOVE ALL "9" TO IREC
               NOT AT END WRITE OREC FROM IREC
               END-READ
           END-PERFORM
       CLOSE IFILE OFILE
       STOP RUN.
```

File Status Codes

The optional FILE STATUS clause specifies a *data-item* that contains a file status code after any I-O statement (except RETURN) is applied to the file. Your program can contain USE procedures that examine the values of such *data-item*s and perform accordingly.

The first digit of a file status code indicates one of the following:

- The I-O operation was successful.
- An AT END condition occurred.
- An INVALID KEY condition occurred.
- A permanent error occurred.
- A logical error occurred.
- An implementation-defined condition occurred.

Table 5-6 lists the I/O statements and shows whether USE procedures, file status codes, or INVALID KEY, AT END, and EOF conditions apply to them.

Table 5-6. I/O Statements and Error Handling that Applies to Them

I/O Statement	USE Procedure	FILE STATUS CODE	INVALID KEY AT END EOF
CLOSE	√	√	
DELETE	√	√	√
EXCLUSIVE	√	√	
OPEN	√	√	
READ	√	√	√
RETURN			√
REWRITE	√	√	√
START	√	√	√
UN-EXCLUSIVE	√	√	
WRITE	√	√	√

Unless more checking is required, your COBOL program should check only the first digit of a file status code. For example, check for an AT END condition by checking that the first digit of the file status code is one. This promotes portability for the following two reasons:

- The program can be compiled on ANSI74 and ANSI85 compilers.
- The program will still work if file status codes are expanded in the future.

More than one file status code can apply to a situation (for example, a program can try to REWRITE a record that is too large and also fail to have a READ before the REWRITE). In this case, the file status code reflects the first error detected. The reason is that when the program detects one error, it does not continue to check for additional errors.

Note The COBOL error message catalog file, COBCAT.PUB.SYS, is updated and released with each compiler version. It contains all the documentation relevant to the compiler and run-time error messages for the matching compiler version, including cause and action information for each error. Following each new update of the product version, you should use the new version of COBCAT.PUB.SYS for this information.

Table 5-8 and Table 5-9 give the file status codes for ANSI85 and ANSI74, respectively. Key terms that appear in those tables are defined below.

Term	Definition
EOF	End of file. The program attempted to read a record following the last record in the file.
AT END	An AT END condition caused a sequential READ statement to fail.
INVALID KEY	One of the categories of file status codes. A code in this category means that an I-O operation failed for one of these reasons: 1. A duplicate key existed. 2. A boundary violation occurred. 3. The record sought was not found. 4. A sequence error occurred. This applies to indexed files only.
Permanent Error	A category of file status code that indicates a problem accessing a permanent file. For example, an I-O statement failed due to an error that precluded further processing of the file.
Logic Error	A category of file status code that indicates a problem with program logic. A code in this category means that an I-O statement failed for one of the following reasons: 1. An improper sequence of I-O statements were performed on the file. 2. A user-defined limit was violated. The record size is an example.

File Status Codes

Table 5-7 groups the ANSI 1985 file status codes by category and explains what each code means for sequential access files, random access files, relative organization files, and indexed file Each entry applies to the columns that it crosses; for example, the information on the file status code 00 applies to all file types.

Table 5-7. ANSI 1985 File Status Codes

File Status Code Category	File Status Code	File Type		
		Sequential Access	Random Access or Relative Organization	Indexed Organization (KSAM)
Successful	00	Successful. No more information available.		
	02	Not applicable.		READ next key value that is the same as current key. WRITE or REWRITE creates duplicate key for alternate key that allows duplicates.
	04	READ length of record does not match file.		
	05	For OPEN, optional file did not exist, so it was created.		
	07	File is not a tape, as the OPEN or CLOSE statement states.	Not applicable.	
AT END	10	READ error. Either EOF, or optional file did not exist.		
	14	Not applicable.	READ error. Record number is too big for relative key data item.	Not applicable.

Table 5-7. ANSI 1985 File Status Codes (continued)

File Status Code Category	File Status Code	File Type		
		Sequential Access	Random Access or Relative Organization	Indexed Organization (KSAM)
INVALID KEY	21	Not applicable.		Sequence error.
	22	Not applicable.	WRITE error. Tried to write a duplicate key (does not apply to REWRITE.)	WRITE or REWRITE error. Tried to write a duplicate key.
	23	Not applicable.	START or READ of missing optional file, or record does not exist.	
	24	Not applicable.	Tried to WRITE beyond file boundary, or sequential WRITE record number is too big for relative key data item.	Tried to WRITE beyond file boundary.
Permanent Error	30	No more information available.		
	31	OPEN, SORT, or MERGE of dynamic file failed due to file name attribute conflict.		
	34	Boundary violation. (Record too big or too small.)	Not applicable.	
	35	OPEN error. Required file does not exist.		
	37	EXTEND or OUTPUT on unwritable file, or I-O operation on file that does not support it, or INPUT on device that is invalid for INPUT.		
	38	OPEN on LOCKed file (LOCKed when last closed).		
	39	OPEN was unsuccessful due to fixed file attribute conflict.		

Table 5-7. ANSI 1985 File Status Codes (continued)

File Status Code Category	File Status Code	File Type		
		Sequential Access	Random Access or Relative Organization	Indexed Organization (KSAM)
Logic Error	41	OPEN on file that is already open.		
	42	CLOSE on file that was not open.		
	43	No READ before REWRITE. READ is required before REWRITE.	No READ before REWRITE or DELETE.	
	44	Boundary violation (Record too big or too small, or rewritten record not the same size.)	Boundary violation (Record too big or too small.)	
	46	READ after AT END or after unsuccessful READ.	READ after AT END or after unsuccessful READ or START.	
	47	READ on file not open for INPUT.	READ or START on file not open for INPUT or I-O.	
	48	WRITE on file not open for OUTPUT or EXTEND.		
	49	REWRITE on file not open for I-O.	REWRITE or DELETE on file not open for I-O.	
Implementation-Defined	9*x*	Unexpected error. The *x* is an ASCII character whose numeric code is an integer between 0 and 255, inclusive, and represents a file system error. For more information, see the MPE XL error message catalog or the *MPE XL Error Message Manual* (Volume 1 or Volume 2).		

Table 5-8 groups the ANSI 1974 file status codes by category and explains what each code means for sequential access files, random access files, relative organization files, and indexed file Each entry applies to the columns that it crosses; for example, the information on the file status code 00 applies to all file types.

Table 5-8. ANSI 1974 File Status Codes

File Status Code Category	File Status Code	File Type		
		Sequential Access	Random Access or Relative Organization	Indexed Organization (KSAM)
Successful	00	Successful. No more information available.		
	02	Not applicable.		READ next key value into current key. WRITE or REWRITE creates duplicate key for alternate key that allows duplicates.
AT END	10	READ error. Either EOF, or optional file did not exist.		
INVALID KEY	21	Not applicable.		Sequence error.
	22	Not applicable.	WRITE or REWRITE error. Tried to write a duplicate key.	
	23	Not applicable.	START or READ of missing optional file, or record does not exist.	
	24	Not applicable.	Tried to write beyond file boundary, or sequential WRITE record number is too big for relative key data item.	Tried to write beyond file boundary.
Permanent Error	30	No more information available.		
	34	Boundary violation. (Record too big or too small.)	Not applicable.	
Implementation-Defined	9*x*	Unexpected error. The *x* is an ASCII character whose numeric code is an integer between 0 and 255, inclusive, and represents a file system error. For more information, see the MPE XL error message catalog or the *MPE XL Error Message Manual* (Volume 1 or Volume 2).		

Table 5-9 compares the ANSI 1985 I-O status codes to their ANSI 1974 equivalents and explains the execution differences between ANSI 1985 and ANSI 1974.

Table 5-9. Differences between ANSI 1985 and ANSI 1974 File Status Codes

ANSI 1985 Status Code	Equivalent ANSI 1974 Status Code	Execution Difference	
		ANSI 1985	ANSI 1974
00	Same.	Not applicable.	
02	Same.	Not applicable.	
04	00	Same.	
05	00	Same.	
07	00	Same.	
10	Same.	Not applicable.	
14	00	READ fails becaust the value of the data item is greater than the PICTURE that describes the key.	READ succeeds.
21	Same.	Not applicable.	
22	Same.	Not applicable.	
23	Same.	Not applicable.	
24	24	Same as for ANSI 1974 except that this code is also returned when the value of the data item is greater then the PICTURE that describes it.	
30	Same.	Not applicable.	
34	None.	Not applicable.	
35	00	The file is not created.	The file is created for an OPEN with the I-O or EXTEND phrase.

Table 5-9.
Differences between ANSI 1985 and ANSI 1974 File Status Codes (continued)

ANSI 1985 Status Code	Equivalent ANSI 1974 Status Code	Execution Difference	
		ANSI 1985	ANSI 1974
37	00	The OPEN fails and a permanent error condition exists for the file.	The OPEN succeeds and the program continues to execute, although it can abort later for another reason.
38	00	The OPEN fails and a permanent error condition exists for the file.	The OPEN fails and a message is printed, even though an error status code is not returned.
39	00	The OPEN fails and a permanent error condition exists for the file.	The OPEN succeeds and may print an error message.
41	9*x*	Same.	
42	9*x*	Same.	
43	00 or 9*x*	DELETE statement fails.	DELETE statement succeeds.
		No difference for REWRITE statement.	
44	00	Statement fails due to logic error.	Statemenet succeeds.
46	10	Statement fails due to logic error.	Continues to return AT END condition or READ ERROR condition.
47	00 or 9*x*	Statement fails due to logic error.	If the file is not open, the status code is 9*x*. If the file is open in the wrong mode, the status code is either 9*x* or 00, and sometime execution continues correctly.
48	00 or 9*x*		
49	00 or 9*x*		

ANSI85 error checking is more stringent than ANSI74 error checking. If you want to use ANSI85 features in your program, but want ANSI74 error checking, use the STAT74 control option. See Chapter 6.

Sequence of Events

If the file status code indicates that the I-O operation was successful, the following occur:

- The NOT AT END or NOT INVALID KEY phrase is executed, if present.
- USE procedures, the AT END phrase, and the INVALID KEY phrase are not executed.
- If the SELECT phrase for the file contains a FILE STATUS phrase, the appropriate file status code is returned.

If the file status code indicates an AT END or INVALID KEY condition, the following occur:

- The NOT AT END or NOT INVALID KEY phrase is not executed.
- The AT END or INVALID KEY phrase is executed, if present.
- If there is a USE procedure, but no AT END or INVALID KEY phrase, the USE procedure is executed.
- If the SELECT clause for the file contains a FILE STATUS clause, the appropriate file status code is returned.
- The program continues to execute after any error procedures (USE, AT END, or INVALID KEY) have been executed.

If the file status code indicates a permanent or logical error or an implementation-defined condition, the following occur:

- If the SELECT clause for the file contains a FILE STATUS clause, the appropriate file status code is returned. The program continues to execute after any USE procedures have been executed.
- If the SELECT clause for the file does not contain a FILE STATUS clause, the program aborts if no applicable USE procedures exist, and a file information display is output. (For an explanation of the file information display, refer to *Using Files: A Guide for New Users of HP 3000 Computer Systems*).
- Any applicable USE procedure is executed.
- INVALID KEY, NOT INVALID KEY, AT END, and NOT AT END phrases are not executed.
- To get more information about the cause of a $9x$ error, call CKERROR to convert x to an MPE XL error number. See also the following example.

Example 1

The following program declares a FILE STATUS item, CHECK-TAPE, and checks it after the READ statement.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT TAPEIN ASSIGN TO "READTAPE"
       FILE STATUS IS CHECK-TAPE.

DATA DIVISION.
FILE SECTION.
FD TAPEIN.
01 TAPE-REC                 PIC X(80).

WORKING-STORAGE SECTION.
77 MPE-ERROR                PIC 9(4) USAGE DISPLAY.
01 CHECK-TAPE.
   02 STAT-KEY-1            PIC X.
   02 STAT-KEY-2            PIC X.
PROCEDURE DIVISION.
      :
    READ TAPEIN   AT END PERFORM NO-MORE-TAPE.
    IF STAT-KEY-1 = "9" THEN
        CALL INTRINSIC "CKERROR" USING CHECK-TAPE MPE-ERROR
        DISPLAY "9 ERROR IN TAPE READ, MPE ERROR IS " MPE-ERROR
        PERFORM ERROR-RTN.
           :
```

Figure 5-3 shows what happens when a run-time I-O error occurs.

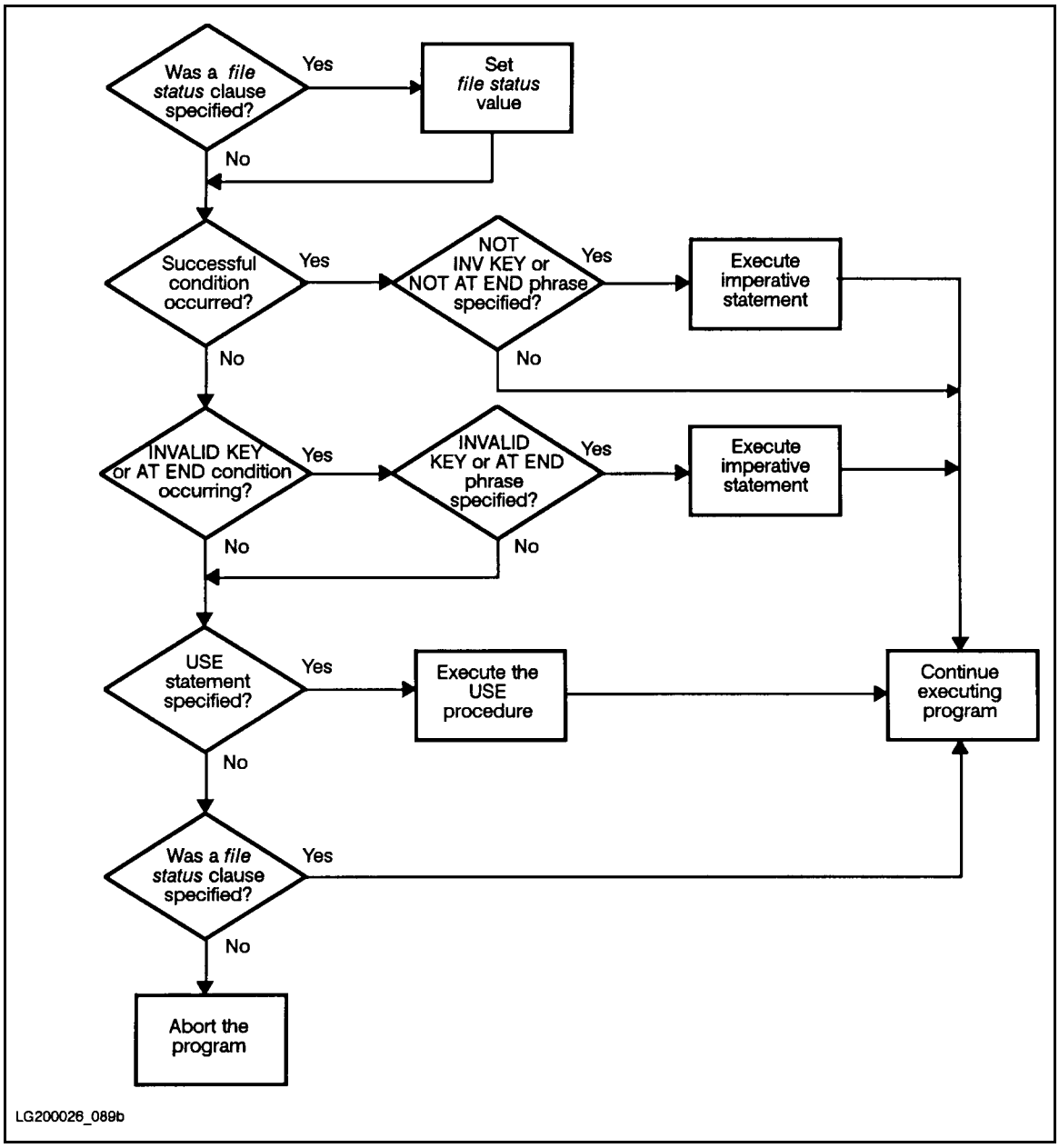


Figure 5-3. Run-Time I-O Error Handling

Example 2

The following program creates a tape file, closes it, then opens it as an input file. Fields in the input records are compared to the values written to ensure that they were processed correctly. The program uses the FILE STATUS clause and a USE statement for error handling.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILE-A ASSIGN TO
    TNAME1
    ORGANIZATION SEQUENTIAL
    ACCESS SEQUENTIAL
    FILE STATUS IS STATUS-FIELD.
DATA DIVISION.
FILE SECTION.
FD  FILE-A.
01  FILE-RECORD.
    02  FILLER  PICTURE  X(120).
WORKING-STORAGE SECTION.
01  STATUS-FIELD.
    02  SFIELD  PIC X.
    02  FILLER  PIC X.
01  EOF-FLAG  PIC 9  VALUE 0.
01  COUNTER  PICTURE S9(9) USAGE COMPUTATIONAL.
01  RECORDS-IN-ERROR PIC S9(5) USAGE COMP VALUE ZERO.
01  ERROR-FLAG PICTURE 9.
01  FILE-RECORD-INFORMATION-REC.
    03 FILE-RECORD-INFO   .
       05 FILE-RECORD-INFO-1.
          07 XFILE-NAME          PIC X(6).
          07 XRECORD-NUMBER      PIC 9(6).
          07 XPROGRAM-NAME       PIC X(6).
          07 RECORDS-IN-FILE     PIC 9(6).
          07 XLABEL-TYPE         PIC X(1).
          07 FILLER              PIC X(95).
```

```
    PROCEDURE DIVISION.
    DECLARATIVES.
    SECT-EXAMPLE-01 SECTION.
        USE AFTER STANDARD EXCEPTION PROCEDURE ON FILE-A.
    TEST-STATUS.
        IF SFIELD EQUAL TO "1"
            MOVE 1 TO EOF-FLAG
            GO TO EXIT-PARA.
        ADD 1 TO RECORDS-IN-ERROR.
    EXIT-PARA.
        EXIT.
    END DECLARATIVES.
    SECT-EXAMPLE-02 SECTION.
    INITIAL-PARA.
        MOVE SPACE TO STATUS-FIELD.
        MOVE "FILE-A" TO XFILE-NAME.
        MOVE "EXAMPLE" TO XPROGRAM-NAME.
        MOVE  500 TO RECORDS-IN-FILE.
        MOVE "S" TO XLABEL-TYPE.
        MOVE 000001 TO XRECORD-NUMBER.
        OPEN OUTPUT FILE-A.
    INIT-FILE.
        MOVE FILE-RECORD-INFO-1  TO FILE-RECORD.
        WRITE FILE-RECORD.
        IF XRECORD-NUMBER  EQUAL TO 500
                GO TO INIT-FILE-EXIT.
        ADD 1 TO XRECORD-NUMBER.
        GO TO INIT-FILE.
    INIT-FILE-EXIT.
        DISPLAY "OPEN WRITE  FILE-A" .
        DISPLAY "FILE CREATED, RECS =" , XRECORD-NUMBER.
        CLOSE FILE-A.
   * A sequential tape file with 120 character records
   * has been created.  The file contains 500 records.
    INIT-READ.
        MOVE ZERO TO COUNTER.
   * This test reads and checks the file created in INIT-FILE.
        OPEN INPUT FILE-A.
```

```
TEST-READ.
    READ FILE-A RECORD.
    IF EOF-FLAG EQUAL TO 1
       GO TO TEST-READ-01.
    MOVE FILE-RECORD TO FILE-RECORD-INFO-1.
    ADD 1 TO COUNTER.
    IF COUNTER GREATER THAN 500
        DISPLAY "MORE THAN 500 RECORDS"
        GO TO FAIL-TEST.
    IF COUNTER NOT EQUAL TO XRECORD-NUMBER
        ADD 1 TO RECORDS-IN-ERROR
        GO TO TEST-READ.
    IF XFILE-NAME  NOT EQUAL TO "FILE-A"
        ADD 1 TO RECORDS-IN-ERROR
        GO TO TEST-READ.
    IF XLABEL-TYPE  NOT EQUAL TO "S"
        ADD 1 TO RECORDS-IN-ERROR.
    GO TO TEST-READ.
TEST-READ-01.
    IF RECORDS-IN-ERROR EQUAL TO ZERO
        GO TO PASS-TEST.
    DISPLAY "ERRORS IN READING FILE-A" .
FAIL-TEST.
    DISPLAY "TEST FAILED".
    DISPLAY "RECORDS IN ERROR =" , RECORDS-IN-ERROR.
    GO TO EXIT-EXAMPLE.
PASS-TEST.
    DISPLAY "TEST PASSED".
    DISPLAY "FILE VERIFIED RECS =" , COUNTER .
EXIT-EXAMPLE.
    CLOSE FILE-A.
    STOP RUN.
```

6

From Program Creation to Program Execution

Introduction

This chapter explains the following:

- How to input your source program to the compiler.
- The control file that the compiler puts in the front of your source file.
- The control options with which you can compile your program.
- How to compile, link, and edit your program.

Source Program Input

You can input your source program to the COBOL compiler in these three ways:

1. In an ASCII file.
2. In an HP TOOLSET TSAM file.
3. One line at a time, through the standard input file, $STDIN (which can be your terminal).

ASCII File

Using an editor, you can create your program in an ASCII file. Then, you can input the ASCII file to the COBOL compiler.

TSAM File

Using HP TOOLSET, you can create your program in a TSAM file.

HP TOOLSET is a software package designed to facilitate three programming activities: coding, compiling, and debugging. It has six key design features:

Feature	Purpose
User Interface	Allows you to communicate with HP TOOLSET with commands and function keys.
On-Line Help Facility	Provides an introduction to HP TOOLSET and an on-line quick reference guide.
Workspace	Controls all files used in development of a single program, including source, listing, object, and program files.
Editor	Allows you to compose and modify your source program on the terminal screen.
Program Translator	Compiles and prepares your program.
Symbolic Debug	Allows you to debug your program.

For complete information on HP TOOLSET, refer to the *HP Toolset/XL Reference Manual*.

$STDIN File

You can input your program one line at a time through the standard input file, $STDIN. The default for $STDIN is your terminal if you are running the compiler interactively. If you are running the compiler from a job stream, the default for $STDIN is the job stream file. In either case, end the source program with ":RETURN".

Example

In this example, your input from the terminal is underlined.

```
:COB85XL

PAGE 0001   HP31500A.00.00  [85] (C) HEWLETT-PACKARD CO. 1987

>       ID DIVISION.
>       PROGRAM-ID. TEST.
>       PROCEDURE DIVISION.
>       P1.
>            DISPLAY "hi mom".
>            STOP RUN.
>:RETURN


0 ERRORS, 0 QUESTIONABLE, 0 WARNINGS

    DATA AREA IS       18 BYTES.
    CPU TIME = 0:00:02.  WALL TIME = 0:01:34.

END OF PROGRAM
```

Control File

When you compile your COBOL program, the compiler includes the control file at the beginning of your source file. The default control file is COBCNTL.PUB.SYS. It contains only comments, but you can add control options to it.

If a file equation that names COBCNTL.PUB.SYS as a formal file designator is in effect when you run the HP COBOL II/XL compiler, the compiler uses that file equation to find the control file to include in your program.

Example

If the following file equation is in effect when you run the HP COBOL II/XL compiler, the compiler uses the file `MYFILE.MYGROUP` as the control file instead of `COBCNTL.PUB.SYS`:

```
:FILE COBCNTL.PUB.SYS=MYFILE.MYGROUP
```

If the compiler cannot open COBCNTL.PUB.SYS or the file that you substituted for it, it issues a warning.

To change the defaults in COBCNTL.PUB.SYS permanently, ask your system manager to change COBCNTL.PUB.SYS. To change them temporarily, use a file equation to redirect COBCNTL.PUB.SYS to another file. The other file must be in standard COBOL format and must contain only preprocessor commands and control options.

Standard COBOL format follows these rules:

Columns	Contains
1-6	Sequence number.
7	One of the following characters:

Character	Means That the Line Contains
␣ (Blank)	COBOL source code
- (hyphen)	Continuation line of COBOL source code.
D	Debug line.
*	Comment.
/	Comment with page eject.
$	Preprocessor command.

8-11	Area A.
12-72	Area B.
73-80	Your own identification field, often the program or source file name. It can be anything, because the compiler copies it to the list file and ignores it. However, for a COBOL copy library, it must be used as *text-name*.

Both the HP COBOL II/XL and HP COBOL II/V compilers use the same default control file (COBCNTL.PUB.SYS). Because some control options are not valid for both compilers, you can create a different control file for use with one of the compilers and issue a file equation to use the second control file, as described in the previous example.

Control Options

You can put control options in any of these places:

- Control file (COBCNTL.PUB.SYS or its substitute).
- COBOL source code.
- RUN command *info* string.

When the source file contains nested or concatenated programs, certain control options must appear at the beginning of the source file. If they appear elsewhere, the compiler issues a warning and ignores them. They need only be turned on once. The control options in this category are:

- CODE
- SYMDEBUG
- VERBS

Control options fall into the following categories:

- Performance options.
- Listing options.
- Debugging options.
- Migration options.
- Standard conformance options.
- Interprogram communication options.
- Miscellaneous options.

Performance Options

Performance options improve the performance of the compiled COBOL program. Currently, OPTIMIZE is the only performance option. Its syntax and effect are:

Performance Option	**Effect**
OPTIMIZE=0	Turns off optimization. This is the default.
OPTIMIZE[=1]	Turns on level one optimization.

The following lists all the possible optimization options:

If you specify:	**You get:**
$CONTROL OPTIMIZE	Level one optimization.
$CONTROL OPTIMIZE=1	Level one optimization.
$CONTROL OPTIMIZE=0	No optimization, the default.
Nothing	No optimization, the default.

Although OPTFEATURES is an interprogram communication option, its settings CALLALIGNED and LINKALIGNED are designed to improve performance. See "Interprogram Communication Options."

Listing Options

Listing options affect the content of the compiler listing. The listing options and their effects are:

Listing Option	Effect
CROSSREF	Prints a cross reference to the list file. The cross reference lists each data and procedure name and the lines that use them. The default is NOCROSSREF.
ERRORS=*n*	Sets the error limit to *n*, an integer. The error limit is the number of errors that the compiler can find before it aborts the program. The default is ERRORS=100.
LINES=*n*	Sets the number of lines per page to *n*. The default is 60 lines.
LIST	Allows actual output from the control options CODE, CROSSREF, MAP, SOURCE, and VERBS to be listed. In included files, use LOCON instead of LIST. This is the default. (Compare to NOLIST, LOCON and LOCOFF.)
LOCOFF	Like NOLIST, except that it can be used in an included file and can be nested. The default is LOCON.
LOCON	Like LIST, except that it can be used in an included file and can be nested. This is the default.
MAP	Prints a map of the program's variables, as offsets from DP, SP, etc. The default is NOMAP.
MIXED	Includes preprocessor commands in the compiler list file. This is the default.
NOCROSSREF	Does not print a cross reference to the list file. This is the default.
NOLIST	Turns off the control options CODE, CROSSREF, MAP, SOURCE, and VERBS. In included files, use LOCOFF instead of NOLIST. The default is LIST.
NOMAP	Does not print a map of the program's variables. This is the default.
NOMIXED	Excludes preprocessor commands from the list file. The default is MIXED.
NOSOURCE	Does not print the source program to the list file. The default is SOURCE.
NOVERBS	Does not print a map of the offsets of the statements. This is the default.
NOWARN	Does not print warning messages to the list file. The default is WARN.
SOURCE	Prints the source program to the list file. This is the default.
VERBS	Prints a map of the offsets of the statements. The default is NOVERBS.
WARN	Prints warning messages to the list file. This is the default.

Debugging Options

Debugging options aid in debugging. The debugging options and their effects are:

Debugging Option	Effect
BOUNDS	Causes the compiler to generate code that checks for values out of bounds at run time. You can use this information for these purposes: 1. To locate values that are out of range in: a. OCCURS DEPENDING ON *identifier* statements. b. Subscripts. c. Indexes. d. Reference modifications. 2. To locate illegal execution of performed paragraphs, specifically: a. Branches out of a performed paragraph. b. Indirectly recursive PERFORM statements. c. Performed paragraphs with common exit points. A limited number of the above three situations will execute as expected before the compiler aborts. BOUNDS provides information that may help you locate the problem. 3. To locate misaligned parameters. 4. To locate unaltered GO TO statements. BOUNDS is not the default.
CHECKSYNTAX	Causes the compiler to include syntax error messages in the compiler listing, but prevents it from generating code. Provides a fast way to check the syntax of your program. This is not the default.
CODE	Dumps an unformatted code listing to the temporary file COBASSM. The default is NOCODE.
DEBUG	Enables CONTROL Y trap in the executable program, allowing program to enter MPE XL System Debugger when the user types CONTROL Y while the program is executing. DEBUG is for main programs only.
NOCODE	Does not dump an unformatted code listing to the file COBASSM. This is the default.
NOVALIDATE	Does not check decimal and packed decimal fields for illegal digits. This is the default. (Compare to VALIDATE.)
SYMDEBUG	Puts symbolic debug information in the executable code. This is not the default.

Debugging Option	Effect
VALIDATE	Checks numeric and packed decimal operands for illegal digits and signs in every arithmetic operation. If an operand is invalid, the program traps to the trap handler, which does what the programmer specified in the COBRUNTIME variable. Whether the program continues after that depends on the value of COBRUNTIME (see "Traps" in Chapter 7). VALIDATE slows execution, so you can omit it if your program gets all its data from computations or from other programs that have checked its validity. VALIDATE is essential, though, if your program gets data from human sources (such as input from a terminal or from a file created by a person) and does not explicitly use class tests. The default is NOVALIDATE.

Note Without the VALIDATE control option, HP COBOL II/XL sometimes treats a blank in a DISPLAY item as if it were a zero and does not trap blanks, but do not depend on this feature. It is implementation dependent, not standard.

Migration Options

Migration options aid in migration (from one machine to another and from one COBOL standard to another). The migration options and their effects are:

Migration Option	Effect
CALLINTRINSIC	Causes the compiler to check all CALL statements against the intrinsic file, SYSINTR.PUB.SYS. If the subprogram is in the intrinsic file, but was not called with the CALL INTRINSIC statement, the compiler issues a warning message and generates code to call the intrinsic.
INDEX16	Specifies 16-bit alignment for index data items. This is the default in Compatibility Mode.
INDEX32	Specifies 32-bit alignment for index data items. This is the default in Native Mode.
QUOTE	Declares the figurative constant QUOTE (").
RLFILE	Specifies that each separately compiled program in a source file goes into its own object module in the RL file. Programs nested within it go into the same object module as it does. See the section "RLFILE" for more information.
RLINIT	Initializes the RL file to empty. See the section "RLINIT" for more information.
STAT74	Causes the I-O functions to operate by the ANSI74 standard rather than the more stringent ANSI85 standard.
SYNC16	Specifies 16-bit alignment for data items that specify SYNC. The default is 32-bit alignment (SYNC32).
SYNC32	Explicitly specifies 32-bit alignment for data items that specify SYNC. This is the default. (Compare to SYNC16.)

RLFILE

The RLFILE control option, with the RLINIT control option, simulates MPE V object module functionality. On the MPE V operating system, programs and subprograms are compiled into Relocatable Binary Modules, or RBMs. RBMs are stored by compilers in USLs (User Segmented Libraries) which are manipulated by the segmenter and the compilers. By using the COBUSL file equation to direct compiler output into a pre-existing USL file you have the ability to do the following:

- Compile several different programs into one USL.
- Replace a single RBM in a USL that contains multiple RBMs, by recompiling a single program.
- Cause the USL file contents to be initialized and cleared with every compilation, by using the USLINIT compiler option.

The RLFILE control option, with the RLINIT control option which the next section describes, gives you on MPE XL the equivalent functionality. On MPE XL, compiler output files are of two types:

- Native Mode Object Files. The file code is NMOBJ.
- Native Mode Relocatable Libraries. The file code is NMRL.

These file types are somewhat similar to RBMs, USLs, and RLs on MPE V.

With the RLFILE option, the compiler creates an NMRL object file. Once main and subprograms have been compiled into this NMRL, the resulting modules can be separately manipulated in the Link Editor or separately replaced in subsequent compilations.

RLFILE operates at the level of the separately compiled program. Concatenated programs (in a single source file) and programs in entirely different source files are separately compiled programs; nested programs are not.

When RLFILE is used, each separately compiled program has a different name, called a *module name*, which is visible to the link editor. Module names are external names (see "External Names" in Chapter 4).

Example 1. This example uses an indirect file list of concatenated programs (COBSRC) to create an NMRL.

```
$CONTROL RLFILE
$INCLUDE MAINP
 END PROGRAM MAIN-P.
$INCLUDE SUB1
 END PROGRAM SUB-1.
$INCLUDE SUB2
 END PROGRAM SUB-2.
```

The following is the command to compile this example:

```
COB85XL COBSRC,COBSRCO
```

COBSRCO is an NMRL because the source file specified RLFILE. A LISTRL command on COBSRCO in the Link Editor will show three separate object modules in COBSRCO, containing the entries *main_p*, *sub_1*, and *sub_2*, respectively. These object modules correspond to the three concatenated programs in COBSRC, and they can be purged or separately manipulated in the Link Editor.

If program SUB-1 has an error, it can be recompiled into COBSRCO with the command:

```
COB85XL SUB1,COBSRCO;INFO="$CONTROL RLFILE"
```

As the result of this command, the module containing the object code from program SUB-1 in COBSRCO is replaced with new object code. No other modules in COBSRCO are changed, because COBSRCO is an NMRL. Thus it is not necessary to recompile all the programs in COBSRC to fix one of them. (If the object code for SUB-1 didn't exist in COBSRCO, then the new code would be appended to COBSRCO in a new module.)

If RLFILE is not specified, whether the compiler output file is an NMRL or an NMOBJ depends on file code of the object file (if it already exists). If the file that COBOBJ refers to already exists and has a file code of NMRL, the object file will be an NMRL. However, since RLFILE was not specified, some RLFILE functionality is not enabled. In particular, concatenated programs, when recompiled, will be put into a single object module within the NMRL. (Recall that if RLFILE is specified, concatenated programs are put into separate object modules in the NMRL, allowing them to be separately manipulated in the Link Editor.)

Example 2. Following the example above, if both programs SUB-1 and SUB-2 have errors, then the following indirect file list, named COBSRC1, would compile just those two programs into the object file COBSRCO.

```
$INCLUDE SUB1
 END PROGRAM SUB-1.
$INCLUDE SUB2
 END PROGRAM SUB-2.
```

The following is the command to compile the above file:

```
COB85XL COBSRC1,COBSRCO
```

RLFILE is not used, but the target file COBSRCO is an NMRL. This means that COBSRCO remains an NMRL, but the code the compiler produces for SUB-1 and SUB-2 appears in a single object module, instead of two. Recall that the first compilation produced an NMRL with three modules, each containing an entry corresponding to MAIN-P, SUB-1, and SUB-2. The second compilation results in an NMRL with two modules. The first module, corresponding to MAIN-P, is unchanged, because the main program was not recompiled. The second two modules, corresponding to SUB-1 and SUB-2, become a single module, containing entries *sub_1* and *sub_2.*

If you want to create NMRLs but don't want to edit your source to add the RLFILE option, simply build an empty file with the code NMRL and direct the output to this file. (This does not allow you to manipulate each program separately in the Link Editor, however. Each compilation produces one module, which can be manipulated.)

RLINIT

The RLINIT control option reinitializes an NMRL object file to empty (similar to USLINIT on MPE V). It has no effect on an NMOBJ object file.

Example. The following example uses COBSRC1 from the previous example:

```
$INCLUDE SUB1
 END PROGRAM SUB-1.
$INCLUDE SUB2
 END PROGRAM SUB-2.
```

The following command compiles COBSRC1:

```
COB85XL COBSRC1,COBSRC0;INFO="$CONTROL RLINIT"
```

The effect of this step depends on the file code of COBSRCO. If COBSRCO is an NMRL, it is reinitialized to empty before compiled code is written to it, and it remains an NMRL. If COBSRCO is an NMOBJ, it remains an NMOBJ. To transform COBSRCO from an NMOBJ to an NMRL, you must purge it and compile with RLINIT or RLFILE.

Table 6-1 and Table 6-2 show how all possible combinations of RLFILE, RLINIT, and the file type of the specified object file affect the actual object file.

Table 6-1 applies when COBOBJ specifies an object file. The file can be $NEWPASS, but it is $NEWPASS by specification rather than by default (see Table 6-2 for the default case).

Table 6-1. RLFILE/RLINIT Functionally With Specified Object File

RLFILE	RLINIT	Specified Object File	Action
ON	ON	Nonexistent. Existing NMRL. Existing NMOBJ. Incorrect file code.[1]	Create NMRL. Rewrite NMRL. Compile-time error. Compile-time error.
ON	OFF	Nonexistent. Existing NMRL. Existing NMOBJ. Incorrect file code.[1]	Create NMRL. Compile into NMRL. Compile-time error. Compile-time error.
OFF	ON	Nonexistent. Existing NMRL. Existing NMOBJ. Incorrect file code.[1]	Create NMRL. Rewrite NMRL. Rewrite NMOBJ. Compile-time error.
OFF	OFF	Nonexistent. Existing NMRL. Existing NMOBJ. Incorrect file code.[1]	Create NMOBJ. Rewrite NMRL.[2] Rewrite NMOBJ. Compile-time error.

1 The file exists, but it is not an NMRL nor an NMOBJ.

2 In this case, not all RLFILE functionality is enabled. See example 2 in the "RLFILE" section.

Table 6-2 applies when no file is specified for COBOBJ and $NEWPASS is the object file.

Table 6-2. RLFILE/RLINIT Functionally With Default File

RLFILE	RLINIT	Specified Object File	Action
ON	ON	Nonexistent Existing NMRL Existing NMOBJ Incorrect file code.[1]	Create NMRL. Rewrite NMRL Create NMRL Create NMRL
ON	OFF	Nonexistent Existing NMRL Existing NMOBJ Incorrect file code.[1]	Create NMRL Compile into NMRL Create NMRL Create NMRL
OFF	ON	Nonexistent Existing NMRL Existing NMOBJ Incorrect file code.[1]	Create NMRL Rewrite NMRL Create NMRL Create NMRL
OFF	OFF	Nonexistent Existing NMRL Existing NMOBJ Incorrect file type.[1]	Create NMOBJ Compile into NMRL Rewrite NMOBJ Create NMOBJ

1 The file exists, but it is not an NMRL nor an NMOBJ.

Standard Conformance Options

Standard conformance options do one of these three things:

- Cause the compiler to generate ANSI standard code in certain cases (ANSISORT and ANSISUB).
- Cause the compiler to flag features that conform to one standard but not another (DIFF74, STDWARN).
- Cancel another standard conformance option (NOSTDWARN).

The standard conformance options and their effects are:

Standard Conformance Option	Effect
ANSISORT	Allows you to open files specified in the GIVING or USING clause of the SORT statement in the input or output procedure of the same SORT statement. This is not the default.
ANSISUB	Maintains values of data items across calls (as SUBPROGRAM does) and allows you to use the CANCEL statement to reset data items to their initial values. The default is that the compilation unit is a main program.
DIFF74	Flags differences between the ANSI 1974 and ANSI 1985 standards. This is not the default.
NOSTDWARN	Does not flag differences between HP COBOL II/XL and Federal Standard COBOL features. See also STDWARN. This option is the default.
STDWARN	Flags differences between HP COBOL II/XL and Federal Standard COBOL (that is, HP extensions are flagged). The possible flags are: HIGH, INT, INTSG, MIN, MINDB, and MINSG. Federal Standard COBOL has three levels: HIGH, INT (intermediate), and MIN (minimal). The default for STDWARN is HIGH. For more information, refer to the *HP COBOL II/XL Reference Manual*. The default is NOSTDWARN.

Interprogram Communication Options

Interprogram communication options make interprogram communication possible. The interprogram communication options and their effects are:

Interprogram Communication Option	Effect
ANSISUB	Maintains values of data items across calls (as SUBPROGRAM does) and allows you to use the CANCEL statement to reset data items to their initial values. The default is that the compilation unit is a main program.
CMCALL	Specifies MPE V conventions for converting external names. This is not the default.
DYNAMIC	Indicates a subprogram with dynamic storage. The INITIAL clause has the same effect. (Compare to SUBPROGRAM and ANSISUB.)
OPTFEATURES=CALLALIGNED[16]	Checks that actual parameters in CALL statements are word-aligned. (for CALLALIGNED) or halfword-aligned (for CALLALIGNED16).
OPTFEATURES=LINKALIGNED[16]	Generates executable code for formal parameters, assuming that actual parameters are word-aligned (for LINKALIGNED) or halfword-aligned (for LINKALIGNED16).
SUBPROGRAM	Indicates a subprogram with its own storage. If your program has a LINKAGE SECTION, this is the default; otherwise, the default is that the compilation unit is a main program. (Compare to DYNAMIC and ANSISUB.)

If your source file contains nested or concatenated programs, you may wish to turn on interprogram communication options in the middle of the file. For example, you may want some subprograms to be ANSISUB, others to be DYNAMIC, and the first program to be the main program.

In this situation, the following rules apply:

- If ANSISUB, DYNAMIC, or INITIAL is turned on for a program, then it also applies to programs nested within that program.
- If ANSISUB, SUBPROGRAM, or DYNAMIC appears in the IDENTIFICATION DIVISION of a program, then it also applies to programs nested within that program.
- If ANSISUB, SUBPROGRAM, or DYNAMIC appears somewhere other than the IDENTIFICATION DIVISION of a program, it applies to the next programs in the source file.

Example

The following shows four programs, A, B, C, and D. ANSISUB applies to programs B and C because C is contained by B, but not to programs A and D. The INITIAL clause in program D's IDENTIFICATION DIVISION has the same effect and scope as DYNAMIC. In this case it only means that program D is DYNAMIC.

Notice that IS INITIAL takes precedence over the control options.

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. A.
    DATA DIVISION.
       :
    PROCEDURE DIVISION.
    BEGIN.
       :
   $CONTROL ANSISUB
    IDENTIFICATION DIVISION.
    PROGRAM-ID. B.
       :
    IDENTIFICATION DIVISION.
    PROGRAM-ID. C.
       :
    END PROGRAM C.
    END PROGRAM B.

    IDENTIFICATION DIVISION.
    PROGRAM-ID. D IS INITIAL.
       :
    END PROGRAM D.
    END PROGRAM A.
```

Miscellaneous Options

Miscellaneous options do not fit into any of the preceding option categories. The miscellaneous options and their effects are:

Miscellaneous Option	Effect
LOCKING	Allows your program to lock all files that are opened during its execution (does not lock the files itself). This is not the default.
USLINIT	This is an HP COBOL II/V option that is ignored by HP COBOL II/XL.
NLS	This option provides support for international (multi-byte or non-ASCII) characters in certain character operations. For a complete description, see the appendix "MPE XL System Dependencies" in the *HP COBOL II/XL Reference Manual*.
POST85	This option enables the built-in COBOL functions, defined in 1989 by Addendum 1 of the ANSI COBOL'85 standard. For a complete description of all the functions and how to call them, see the chapter "COBOL Functions" in the *HP COBOL II/XL Reference Manual*.

Compiling, Linking, and Executing Your Program

When you compile your program, the COBOL compiler translates your COBOL source program to object code and resolves calls to nested programs.

When you link your program, the link editor resolves calls to subprograms that are to be bound at link time (not all subprograms; see Chapter 4). Your program must be linked whether it calls such subprograms or not. The link editor uses object code to produce a program file.

When you execute your program, the loader loads the program file into memory and the operating system executes it.

Figure 6-1 shows how a source program becomes an executing program.

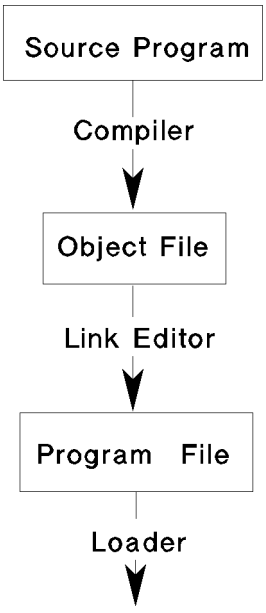


Figure 6-1. How a Source Program Becomes an Executing Program

If the source program consists of a main program and one or more subprograms, the main program and each subprogram must be compiled separately. The resulting object files must be linked together into a single program file. An object file cannot contain more than one program.

Figure 6-2 shows the input to the COBOL compiler and the output from it. Note that COPYLIB and INCLUDE files can also be used for input. Formal file designators are in italic capital letters.

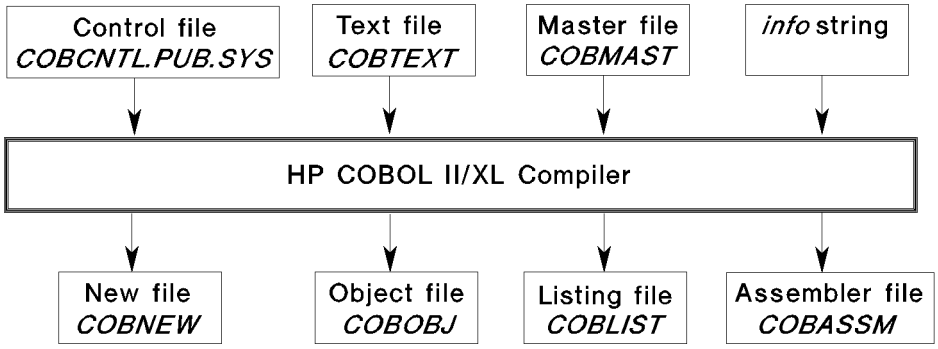


Figure 6-2. COBOL Compiler Input and Output

The rest of this section explains the following:

- The two compiler entry points and the two compiler modes.
- How to use file equations with the FILE command.
- How to run the COBOL compiler with the MPE XL RUN command.
- How to compile, link, and execute your program with Native Mode command files.
- How to compile, link, and execute your program with Compatibility Mode compiler UDCs or commands.
- How to use relocatable and executable libraries.
- Examples of compiling, linking, and executing programs.

Compiler Entry Points and Modes

The COBOL compiler is "two compilers in one." That is, it has two entry points, ANSI74 and ANSI85. When you invoke it through the ANSI74 entry point, ANSI COBOL 1974 syntax and semantics apply. When you invoke it through the ANSI85 entry point, ANSI COBOL 1985 syntax and semantics apply. The ANSI74 entry point is provided for compatibility with older COBOL compilers.

The COBOL compiler also has two modes, Native Mode and Compatibility Mode. In Native Mode, the compiler produces object code designed especially for the MPE XL operating system. This code runs very fast on MPE XL. In Compatibility Mode, the compiler produces object code designed especially for the MPE V operating system. It runs on MPE XL, but not as efficiently as Native Mode object code does, because MPE XL must simulate the MPE V system hardware and microcode. Compatibility Mode is provided for compatibility with the MPE V operating system.

Both entry points are available in both modes. In summary, the entry points differ in the source code that they recognize, and the modes differ in the object code that they generate.

Table 6-3 shows what the compiler does with each entry point in Native Mode and Compatibility Mode.

Table 6-3. Entry Point and Mode Combination

Entry Point	Native Mode	Compatibility Mode
ANSI85	Translates ANSI COBOL 1985 source code to MPE XL object code.	Translates ANSI COBOL 1985 source code to MPE V object code.
ANSI74	Translates ANSI COBOL 1974 source code to MPE XL object code.	Translates ANSI COBOL 1974 source code to MPE V object code.

File Equations

When you use a file equation (the FILE command), its formal designator must match the file name in the ASSIGN clause of the SELECT statement.

Example

The following SELECT statement declares the logical file MY-FILE:

```
SELECT MY-FILE ASSIGN TO "MFILE".
```

The following file equation associates the logical file MY-FILE with the physical file REALFILE:

```
:FILE MFILE=REALFILE
```

You can use a file equation to do the following:

- Tell your COBOL program to change the temporary file that it creates to a permanent file. For example, the following FILE command causes the temporary file MFILE to be made a permanent file:

  ```
  :FILE MFILE;SAVE
  ```

  One alternative to the FILE command is to use the SAVE command SAVE MFILE before the job or session ends. For example, the following also causes the temporary file MFILE to be made a permanent file:

  ```
  :SAVE MFILE
  ```

- Override default file characteristics.
- Create a circular file.
- Create a message file.

An alternative to a file equation or the FILE command in each of the above cases is to create the file with the BUILD command before you execute the program. For more information on files and the FILE command, see Chapter 5.

Native Mode Compiler Command Files and RUN Command

You can invoke the Native Mode COBOL compiler and compile your COBOL program in Native Mode with either the RUN command or one of the six command files COB85XL, COB85XLK, COB85XLG, COB74XL, COB74XLK, or COB74XLG. These command files can compile, link, and execute your program. You can use one of the six existing command files or write your own. For more information on compiling, linking, and executing your program, see the appendix "MPE XL System Dependencies" in the *HP COBOL II/XL Reference Manual*.

Compatibility Mode Compiler UDCs and Commands

To compile your program in Compatibility Mode, use a UDC (User-Defined Command) or a command to compile, link, and execute your program. Table 6-4 and Table 6-5 list each UDC and command that invokes the Compatibility Mode COBOL compiler, the entry point through which it invokes the compiler, and its other effects.

Table 6-4. Compatibility Mode UDCs

UDC	Entry Point	Invokes the compiler and:
COBOLIIX	ANSI85	Creates an object file.
COBOLIIXPREP		Links the object file and creates a program file.
COBOLIIXGO		Creates a program file in $OLDPASS and runs it.

Table 6-5. Compatibility Mode Commands

Command	Entry Point	Invokes the compiler and:
COBOLII	ANSI74	Creates an object file.
COBOLIIPREP		Links the object file and creates a program file.
COBOLIIGO		Creates a program file in $OLDPASS and runs it.

Syntax of UDCs

```
COBOLIIX [textfile][,[uslfile[,[listfile][,[masterfile]
         [,[newfile]]]]]][,info]

COBOLIIXPREP [textfile][,[progfile[,[listfile][,[masterfile]
             [,[newfile]]]]]][,info]

COBOLIIXGO [textfile][,[listfile][,[masterfile] [,[newfile]]]][,info]
```

Syntax of Commands

```
COBOLII [textfile][,[uslfile[,[listfile][,[masterfile] [,[newfile]]]]]]
        [;INFO=info][;WKSP=workspacename]

COBOLIIPREP [textfile][,[progfile[,[listfile][,[masterfile]
            [,[newfile]]]]]][;INFO=info][;WKSP=workspacename]

COBOLIIGO [textfile][,[listfile][,[masterfile][,[newfile]]]]
          [;INFO=info][WKSP=workspacename]
```

Parameters

textfile — MPE or TSAM file containing your source program. This file can be compiled. The formal designator is COBTEXT. The default is $STDIN.

uslfile — Relocatable object code file. This file can be linked. The formal designator is COBUSL. The default is $OLDPASS.

progfile — Executable program file. This file can be executed. The default is $NEWPASS.

listfile — MPE file on which your source code will be listed. The formal designator is COBLIST. The default is $STDLIST.

masterfile — MPE or TSAM file to be merged with *textfile* to produce a composite source program. The formal designator is COBMAST. If *masterfile* is omitted, the entire source is from *textfile.*

newfile — MPE file into which the merged *textfile* and *masterfile* is written. For details, refer to the *HP COBOL II/XL Reference Manual.* The formal designator is COBNEW. The default is that no new file is written.

info — A string whose value is a command list of the form:

"$*compiler_command*[$*compiler_command*] . . ."

where no *compiler_command* contains the character $ (even if it is within quotes). Refer to the *HP COBOL II/XL Reference Manual* for more information on compiler commands.

If the number of commands is long enough, you can use an ampersand (&) to continue the *info* string. The length limit for a compiler command is the same as the length limit for a source program line.

In the listing file, the string "INFO=" appears where the sequence numbers normally appear.

The *info* string is processed before any source, including compiler commands in the source. Therefore, you may not want to use the default settings of these commands in the source file. You should only include commands such as SUBPROGRAM, which are required for proper compilation, in the source file. This allows you to specify commands like NOLIST, MAP, BOUNDS, or CROSSREF uniquely within the *info* string for each compilation.

workspacename — Work space in which HP TOOLSET/XL can manage versions of the source program.

Libraries

Your program can use subprograms from relocatable libraries and executable libraries. Relocatable and executable libraries differ in content and in when they are referenced. The following table lists these differences:

Table 6-6. Differences between Relocatable and Executable Libraries

Library	Contents	Referenced At
Relocatable (RL)	Relocatable object modules.	Link time.
Executable (XL)	Executable object modules.	Run time.

Relocatable Libraries

A relocatable library contains relocatable object modules and is referenced at link time.

Example. The following shows how you can use the Link Editor to put the subprogram SUB1 into the relocatable library RLLIB. Your input is underlined:

```
:LINKEDIT
LinkEd>RL RLLIB
LinkEd>ADDRL FROM=SUB1
LinkEd>EXIT
```

The following command links subprograms from RLLIB with the object file of the main program, TMAIN, and creates the program file TMAINP:

```
:LINK TMAIN,TMAINP;RL=RLLIB
```

Executable Libraries

An executable library contains executable object modules and is referenced at run time.

The three ways to tell the operating system which executable library contains the subprograms for your program are:

1. Specify the library in the LINK command.

   The advantage of this method is that you need not remember to specify the library in the RUN command each time you execute your program. The only time you cannot use this method is when you want to execute the program with different executable libraries at different times.

   The following command links subprograms from XLLIB.GROUP.ACCOUNT with the object file of the main program, TMAIN, and creates the program file TMAINP:

   ```
   :LINK TMAIN,TMAINP;XL=XLLIB.GROUP.ACCOUNT
   ```

   Either of the following commands runs the program:

   ```
   :RUN TMAINP      This uses the RUN command.
   :TMAINP          This uses the implied RUN command.
   ```

2. Specify the library in the RUN command.

   The advantage of this method is that you can execute the program with different executable libraries at different times.

   The following command executes the program TMAIN, in the program file TMAINP, with the executable library XLLIB.GROUP.ACCOUNT:

   ```
   :RUN TMAINP;XL="XLLIB.GROUP.ACCOUNT"
   ```

3. Name the library XL. In the RUN command, specify LIB=G if the library is in your group, LIB=P if the library is in your account, or LIB=S if the library is NL.PUB.SYS or XL.PUB.SYS. LIB=S is the default.

   This is the least flexible method. It is similar to the method used on the MPE V operating system.

Examples of Compiling, Linking, and Executing Programs

This section contains several examples of compiling, linking and executing programs.

Example 1

The first example compiles, links, and executes a program. Either the program calls no subprograms, or the subprograms that it calls are in the executable library NL.PUB.SYS or XL.PUB.SYS.

```
:COB85XL TPROG,,$NULL
:LINK
:RUN $OLDPASS
```

The first command above compiles the COBOL source program, TPROG, into the default object file, $OLDPASS. $NULL specifies no listing.

The second command links the default object file, $OLDPASS, into the default program file, $NEWPASS, which becomes $OLDPASS when it is closed.

The third command executes the program file, $OLDPASS.

The following single command is equivalent to the above sequence of three commands:

```
:COB85XLG TPROG,$NULL
```

Example 2

The second example compiles, links, and executes a main program and two subprograms. The main program and each subprogram is in a separate object file. The keyword RUN in the last command is optional.

```
:COB85XL TMAIN,TMAIN0
:COB85XL SUB1,SUB10
:COB85XL SUB2,SUB20
:LINK FROM=TMAIN0,SUB10,SUB20;TO=TMAINP
:RUN TMAINP
```

Example 3

The third example compiles, links, and runs concatenated programs. The first program is a main program. It uses an indirect file list of the programs to be compiled. When programs are concatenated, the compiler is invoked once for all of them, and the compiler output for these programs is in a single object file, so the FROM list in the LINK command only needs to specify one object file. The default $OLDPASS is used below.

The "indirect" file, COBSRC, contains:

```
$INCLUDE MAINSRC
 END PROGRAM MAINSRC.
$INCLUDE SUB1SRC
 END PROGRAM SUB1SRC.
$INCLUDE SUB2SRC
 END PROGRAM SUB2SRC.
$INCLUDE SUB3SRC
```

The END PROGRAM header is not necessary for the last concatenated program in a file. This is an easy way to migrate your current main and subprograms to concatenated programs, and simplify your compiles and links. The following commands compile and link the file COBSRC:

```
:COB85XL COBSRC
:LINK
:$OLDPASS
```

The defaults for the object file and program file are used above. The RUN keyword is not necessary to run the program file, $OLDPASS.

7

Debugging Your Program

Introduction

This chapter does the following:

- Lists the control options that aid debugging (explained in Chapter 6.)
- Explains how to use the compiler listing and link map in debugging your program.
- Lists messages issued at compile time and run time.
- Describes Debug, the MPE XL System Debugger.
- Describes the symbolic debuggers (HP Symbolic Debugger/XL and HP TOOLSET/XL debugger).
- Lists compiler limits and how to work around some of them.

In this chapter, underlining sets your input apart from the prompt or output that is shown with it.

The material in this chapter applies only to Native Mode programs.

Control Options for Debugging

The following control options aid debugging. For more information on them, see Chapter 6. For an example of a compiler listing of a program with these options, see "Maps Example."

Table 7-1. Debugging Control Options

BOUNDS	DEBUG	SYMDEBUG	VERBS
CODE	MAP	VALIDATE	

Compiler Listing

The compiler listing that the compiler produces when you specify the control options MAP and VERBS provides most of the information that you need to debug your program. The remainder of the information is supplied by the link map. For information on the MAP and VERBS control options, see Chapter 6. For information on the link map, see "Link Map," under "Using Debug."

The following is a compiler listing for a program that includes the control options MAP, VERBS, and CROSSREF. Explanations for the **bold**, bracketed numbers follow the listing.

```
                                        [1]      [2]               [3]
PAGE 0001   HEWLETT-PACKARD 31500A.00.00  [85] COBOL II/XL THU, MAR 12, 1987,
11:23 AM (C) HEWLETT-PACKARD CO. 1987
             [4]

 00001 COBCNTL  001000*    COBCNTL.PUB.SYS  Defaults are:                        \
 00002 COBCNTL  002000*CONTROL LIST,SOURCE,NOCODE,NOCROSSREF,ERRORS=100,NOVERBS,   |
                       WARN                                                      | [5]
 00003 COBCNTL  003000*CONTROL LINES=60,NOMAP,MIXED,QUOTE=",NOSTDWARN,SYNC32,     |
                       INDEX32                                                   /
 00004          004700$control ansisort

 00005           INFO=$control dynamic,verbs,crossref,map [6]
 00007          001000 IDENTIFICATION DIVISION.
 00008          002000 PROGRAM-ID. DATEFORMAT.
 00009          003000 ENVIRONMENT DIVISION.
 00010          004000 DATA DIVISION.
 00011          005000 WORKING-STORAGE SECTION.
 00012          006000 77 PGM-ID               PIC X(10) VALUE 'DATEFORMAT'.
 00013          007000 01 CHAR-CNT                 PIC S9(4) BINARY.
 00014          008000 01 YEAR-4                   PIC 9999.
 00015          009000 01 MONTH-NAMES.
 00016          010000    05                       PIC X(10) VALUE "JANUARY".
 00017          011000    05                       PIC X(10) VALUE "FEBRUARY".
 00018          012000    05                       PIC X(10) VALUE "MARCH".
 00019          013000    05                       PIC X(10) VALUE "APRIL".
 00020          014000    05                       PIC X(10) VALUE "MAY".
 00021          015000    05                       PIC X(10) VALUE "JUNE".
 00022          016000    05                       PIC X(10) VALUE "JULY".
 00023          017000    05                       PIC X(10) VALUE "AUGUST".
 00024          018000    05                       PIC X(10) VALUE "SEPTEMBER".
 00025          019000    05                       PIC X(10) VALUE "OCTOBER".
 00026          020000    05                       PIC X(10) VALUE "NOVEMBER".
 00027          021000    05                       PIC X(10) VALUE "DECEMBER".
 00028          022000 01 MONTH-TABLE  REDEFINES MONTH-NAMES.
 00029          023000    05   MONTH-NAME          PIC X(10) OCCURS 12 TIMES
 00030          024000                                       INDEXED BY I.
 00031          025000 LINKAGE SECTION.
 00032          026000 01 DATE-IN.
 00033          027000     05  YY-IN               PIC 99.
 00034          028000     05  MM-IN               PIC 99.
 00035          029000     05  DD-IN               PIC 99.
 00036         +030000 01 DATE-OUT                 PIC X(30).
  [7] [8]     [9] [10] [11]
```

```
00037          031000 PROCEDURE DIVISION USING DATE-IN DATE-OUT.
00038          032000 STARTA.
00039          033000     display "date-format" date-in
00040          034000     SET I TO MM-IN
00041          035000     ADD 1900 TO YY-IN GIVING YEAR-4
00042          036000     MOVE 1 TO CHAR-CNT
00043          037000     STRING MONTH-NAME(I) DELIMITED BY " "
00044          038000            INTO DATE-OUT
00045          039000            WITH POINTER CHAR-CNT
00046          040000            END-STRING
00047          041000     STRING  " " DD-IN ", " YEAR-4 DELIMITED BY SIZE
00048          042000            INTO DATE-OUT
00049          043000            WITH POINTER CHAR-CNT
00050          044000            END-STRING
00051          045000     MOVE SPACES TO DATE-OUT(CHAR-CNT:).
00052          046000
00053          047000 EXIT-PARA.
00054          048000     EXIT PROGRAM.

                        [12]
PAGE 0002/COBTEXT  DATEFORMAT      SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME      BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ
                                                                    [25][26]
 WORKING-STORAGE SECTION

 00012 77  PGM-ID           SP       -114         A DISP      AN
 00013 01  CHAR-CNT         SP       -108         2 COMP      NS
 00014 01  YEAR-4           SP       -104         4 DISP      N
 00015 01  MONTH-NAMES      SP       -100        78 DISP      AN
 00016 05   FILLER          SP       -100         A DISP      AN
 00017 05   FILLER          SP        -F6         A DISP      AN
 00018 05   FILLER          SP        -EC         A DISP      AN
 00019 05   FILLER          SP        -E2         A DISP      AN
 00020 05   FILLER          SP        -D8         A DISP      AN
 00021 05   FILLER          SP        -CE         A DISP      AN
 00022 05   FILLER          SP        -C4         A DISP      AN
 00023 05   FILLER          SP        -BA         A DISP      AN
 00024 05   FILLER          SP        -B0         A DISP      AN
 00025 05   FILLER          SP        -A6         A DISP      AN
 00026 05   FILLER          SP        -9C         A DISP      AN
 00027 05   FILLER          SP        -92         A DISP      AN
 00028 01  MONTH-TABLE      SP       -100        78 DISP      AN       R [23]
 00029 05   MONTH-NAME      SP       -100         A DISP      AN         O [24]
           I               SP       -11C         4 INDEX NAME

 LINKAGE SECTION

 00032 01  DATE-IN          P 00        0         6 DISP      AN
 00033 05   YY-IN           P 00        0         2 DISP      N
 00034 05   MM-IN           P 00        2         2 DISP      N
 00035 05   DD-IN           P 00        4         2 DISP      N
 00036 01  DATE-OUT         P 01        0        1E DISP      AN
 [13]  [14] [15]            [16]       [17][18]  [19][20]  [21] [22]
```

Note The symbol table map on PAGE 0002 is due to the control option MAP. The address of each data name in the program is specified in terms of base and offset to that base.

```
                          [12]
PAGE 0003/COBTEXT  DATEFORMAT      SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME      BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ

 STORAGE LAYOUT   (#ENTRYS)

   FIRST TIME FLAG          SP       -130        4 \
   RUN TIME $ . ,           SP       -124        4  |
   SORT/MERGE PLABEL        SP       -120        4  |
   INDEX TABLE         (1)  SP       -11C        4  | [27]
   TALLY                    SP       -118        4  |
   USER STORAGE             SP       -118       90  |
   PARAMETER POINTERS  (2)  SP        -88        8  |
   TEMPCELL pool            SP        -80       44 /
                       [28]
```

Note The symbol table map on PAGE 0003 is due to the control option MAP. The address of each data name in the program is specified in terms of base and offset to that base.

```
PAGE 0004/COBTEXT  DATEFORMAT      STATEMENT OFFSETS
                                   Entry = dateformat [29]

  STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

    38       84     40       BC     42      188     50      214     53   304

    39       84     41      114     46      194     51      2D0     54   308
  [30]       [31]
```

Note The verb map on PAGE 0004 is due to the control option VERBS.

```
  PAGE 0005/COBTEXT  DATEFORMAT      CROSS REFERENCE LISTING
                                          IDENTIFIERS
CHAR-CNT                           00013   00042   00045   00049   00051
DATE-IN                            00032   00037   00039
DATE-OUT                           00036   00037   00044   00048   00051
DD-IN                              00035   00047
I                                  00030   00040   00043
MM-IN                              00034   00040
MONTH-NAME                         00029   00043
MONTH-NAMES                        00015   00028
MONTH-TABLE                        00028
PGM-ID                             00012
YEAR-4                             00014   00041   00047
YY-IN                              00033   00041
[32]                                [33]


 PAGE 0006/COBTEXT  DATEFORMAT      CROSS REFERENCE LISTING
                                          PROCEDURES
EXIT-PARA                          00053
STARTA                             00038



PAGE 0007/COBTEXT  DATEFORMAT      COBOL ERRORS and WARNINGS [34]
COBOL ERRORS:

LINE #   SEQ #  COL  ERROR  SEV   TEXT OF MESSAGE
------  ------  ---  -----  ---   ---------------

 00046  040000  80    055    W    LEFT TRUNCATION MAY OCCUR.
 00050  044000  80    055    W    LEFT TRUNCATION MAY OCCUR.

 [35]    [36]   [37]   [38]   [39]  [40]

0 ERRORS, 0 QUESTIONABLE, 2 WARNINGS

    DATA AREA IS       F4 BYTES. [41]
    CPU TIME = 0:00:12.  WALL TIME = 0:00:29.
               [42]                 [43]
```

Note The cross reference on PAGE 0005 and PAGE 0006 is due to the control option CROSSREF. It lists the section, paragraph, and program names separate from the identifiers and macros.

The bracketed numbers on the preceding compiler listing have these meanings:

Number	Meaning of Listing
[1]	Version number of the compiler. Use this number when communicating with Hewlett-Packard about this compiler.
[2]	Entry point (85 for ANSI85, 74 for ANSI74).
[3]	Compilation date.
[4]	Copyright for compiler.
[5]	Control file (COBCNTL.PUB.SYS here).
[6]	*info* string.
[7]	Compiler-generated listing line number (also called statement number).
[8]	Columns 73 through 80 of the source file. If the source contains a copy module from a COPYLIB, these columns contain the module name. Anything in columns 73 through 80 appears on the compiler listing. In a macro or include file, it is good programming practice to put the name of the macro or included file in columns 73 through 80, so that the compiler listing will identify those source lines as being from that macro or included file.
[9]	If both COBMAST and COBTEXT are specified, a plus (+) appears in this column of each line of the file COBTEXT.
[10]	Columns one through six of the source file (source sequence number).
[11]	Columns seven through 72 of the source file (COBOL source code).
[12]	PROGRAM-ID.
[13]	Compiler-generated listing line number (same as [7]).
[14]	Level number.
[15]	Data name.
[16]	Base. Possible bases are:

Base	Use
DP	Main program storage.
SP	Subprograms with dynamic storage (control option DYNAMIC or INITIAL clause in the PROGRAM-ID paragraph).
OWN	Subprograms with static storage (control option ANSISUB or SUBPROGRAM).
P*nn*	Parameter number *nn*.
EXT	External item.
CODE	Literals used in program.

[17] Offset from base (in hexadecimal, positive or negative).

[18] An asterisk after an offset indicates non-optimal alignment of data name.

[19] Size in bytes (hexadecimal).

[20] USAGE. Terms and meanings are:

Term	Meaning
DISP	DISPLAY.
COMP	COMPUTATIONAL or BINARY.
COMP-SYNC	COMPUTATIONAL SYNCHRONIZED or BINARY SYNCHRONIZED.
COMP-3	COMPUTATIONAL-3 or PACKED-DECIMAL.
INDEX	Index (in USAGE IS INDEX).
INDEX NAME	Index name (in an INDEXED BY clause).
SEQUENTIAL	Organization is sequential, access is sequential.
RANDOM	Access is random.
RELATIVE SEQUENTIAL	Organization is relative, access is sequential.
RELATIVE RANDOM	Organization is relative, access is random.
RELATIVE DYNAMIC	Organization is relative, access is dynamic.
INDEXED SEQUENTIAL	Organization is indexed, access is sequential.
INDEXED RANDOM	Organization is indexed, access is random.
INDEXED DYNAMIC	Organization is indexed, access is dynamic.

[21] Asterisk before category indicates unsigned binary or packed decimal item (arithmetic is faster with signed numbers than unsigned numbers).

[22] Category. The categories and their meanings are:

Category	Meaning
N	Numeric
A	Alphabetic
AN	Alphanumeric
NE	Numeric edited
ANE	Alphanumeric edited
NS	Signed numeric, sign trailing
SN	Signed numeric, sign leading
NS-SEP	Signed numeric; sign trailing, separate
SEP-SN	Signed numeric; sign leading, separate

[23] R indicates REDEFINES clause.

[24] O indicates OCCURS clause.

[25] J indicates JUSTIFIED clause.

[26] BZ indicates BLANK WHEN ZERO clause.

[27] Internal compiler data. Only the PARAMETER POINTERS are useful to you. It gives the address of the addresses of the subprogram parameters.

[28] Number of parameters, files, or index names (decimal).

[29] Offsets in this map are offsets to this location. This is the PROGRAM-ID name or chunk name.

[30] Compiler-generated listing line number (same as [7]).

[31] Offset from entry or chunk (hexadecimal).

[32] Name (possibly qualified).

[33] Listing line numbers that reference the name (the first number is the line on which the name is declared).

[34] Error message listing.

[35] Approximate listing line number of error.

[36] Approximate source sequence number of error.

[37] Approximate column location of error (column 80 if the error was detected during code generation).

[38] Error number.

[39] Severity of error.

[40] Error message text.

[41] Approximate total data area size (hexadecimal).

[42] CPU time required to compile the program (*hh:mm:ss*).

[43] Elapsed time required to compile the program (*hh:mm:ss*).

Messages

This section explains the messages that help you debug your program. They are of two types, compile-time messages and run-time error messages. Compile-time messages are issued by the compiler, and are not always associated with errors. Run-time error messages are issued by the COBOL run-time library, and are always associated with errors.

Text for both types of errors comes from the file COBCAT.PUB.SYS (COBOL CATalog). Beneath the text of each error message, COBCAT.PUB.SYS has an explanation of the message. Each line of the explanation begins with a dollar sign ($). For instructions for printing COBCAT.PUB.SYS, refer to the *HP COBOL II/XL Reference Manual*.

For run-time error processing, the COBOL run-time library accesses the COBMAC.PUB.SYS (COBOL MACro) file, as well as COBCAT.PUB.SYS. The COBMAC file is used to provide additional information when a trap is detected.

Compile-Time Messages

Compile-time messages are issued by the compiler. They are of six severities: warning, questionable, serious, disastrous, nonstandard, and informational. The severity of a message determines its effect on the compiler—whether it continues to compile, whether it generates code, and whether the code executes correctly.

When the compiler issues a message, it sets the Job Control Word (JCW). You can have your job stream compile your program and then check the JCW. If the JCW setting is FATAL (in which case the program did not compile successfully), your job stream will end, rather than try to link and execute the program.

For each class of compile-time message, Table 7-2 gives the message number range, the JCW setting, an explanation, and advice.

Table 7-2. Compile-Time Message Severities

Message Severity	Message Number Range	JCW Setting	Explanation	Advice
Warning (W)	1-99	Not set.	The compiler generated code for the program, but it will not execute correctly in the "worst case."	If the "worst case" could happen, change the code so that it will execute correctly in that case.
Questionable Error (Q)	100-399	WARN	The compiler generated code for the program, but it will probably not execute correctly.	Change the program to eliminate this error.
Serious Error (S)	400-449	FATAL	The compiler could not generate code for the program.	Change the program to eliminate this error.
Disastrous Error (D)	450-499	FATAL	The compiler could not generate code for the program and cannot continue to compile it. The compiler listing includes a stack dump. (Most disastrous errors are caused by file errors from the files that the compiler accesses; for example, when the compiler cannot find the file that COBTEXT references.)	Check the spellings of the file names in the file equations and the commands that invoked the compiler.
Nonstandard Warning (N)	500-539	Not set	The compiler generated code for the program, but it may not execute correctly on non-HP computers. A nonstandard warning flags an HP extension to standard COBOL or a standard COBOL feature that is above the level that the control option STDWARN specifies.	If the program is to run on non-HP computers, change the program to eliminate this warning.
Informational Messages (I)	900-999	Not set	Usually caused by other errors in the program.	Correct the other errors in the program and recompile.

Run-Time Error Messages

Run-time error messages are numbered from 500 to 755 and are issued by the HP COBOL II/XL run-time library. Unlike compile-time messages, they are always associated with errors. The errors fall into two classes: input-output errors and traps. This section explains input-output errors and data validation, a related issue. This section only briefly describes traps. The *HP COBOL II/XL Reference Manual* explains traps and how to handle them. The section "Debugging Trap Errors" later in this chapter has example programs that illustrate these errors.

Input-Output Errors

Input-output errors cause most run-time error messages. When an input-output error occurs, the following happens:

1. The error message is printed.
2. If the error is file-related (most are), the file system error number and message are printed and the intrinsic PRINTFILEINFO executes, displaying file information. If the error is not file-related, only the error message is printed.
3. If the program contains a FILE STATUS clause, INVALID KEY phrase, AT END phrase, or USE procedure (that is, if the program can detect the error), then execution continues without printing any error message or file information; otherwise, the program aborts (see Figure 5-3).

Some of the COBOL functions call routines in the Compiler Library or FORTRAN library. If a Compiler Library or FORTRAN routine detects an error, the Compiler Library will output an error message. These error messages are documented in the *Compiler Library/XL Reference Manual*.

Run-Time Traps

A *run-time trap* is an interruption of the flow of program control, caused by an exception condition. After a trap-handling routine executes, the program can sometimes be restarted, depending on how you set the environment variable COBRUNTIME.

The COBOL compiler supports the following traps:

- Illegal Decimal Digit (Error 710).
- Illegal ASCII Digit (Error 711).
- Bad Parameter (Error 745)
- No Exception Phrase on CALL (Error 746)
- No SIZE ERROR phrase (Error 747)
- Paragraph Stack Overflow (Error 748).
- Subscript, Index, Reference Modifier, or DEP-ON Bounds Error (Error 751).
- Address Alignment (Error 753).
- Invalid GO TO (Error 754).

For traps to do anything other than abort the program, you must compile your program with $CONTROL VALIDATE and $CONTROL BOUNDS, and set the global variable COBRUNTIME before you run your program.

For a complete discussion of these traps and COBRUNTIME, see the appendix "MPE XL System Dependencies" in the *HP COBOL II/XL Reference Manual*.

For example programs that illustrate these traps, see "Debugging Trap Errors" later in this chapter.

Data Validation

When you use the control option VALIDATE and the run-time error handling option M or N in columm 1 of COBRUNTIME, and an illegal ASCII or decimal digit is encountered, in most cases the trap handler changes the source field itself, rather than a copy of it. That is, the source field is also the target field.

Table 7-3 gives the valid ASCII digits with which invalid unsigned ASCII digits are replaced if you specify the run-time error handling option M or N. Any invalid unsigned ASCII digit that does not appear in Table 7-3 is replaced by zero.

Table 7-3. Valid Replacements for Invalid Unsigned ASCII Digits

Invalid Unsigned ASCII Digit	Valid Replacement Digit
A, a, J, j, /	1
B, b, K, k, S, s	2
C, c, L, l, T, t	3
D, d, M, m, U, u	4
E, e, N, n, V, v	5
F, f, O, o, W, w	6
G, g, P, p, X, x	7
H, h, Q, q, Y, y	8
I, i, R, r, Z, z	9

Table 7-4 gives the valid ASCII digits with which invalid signed ASCII digits are replaced if you specify the run-time error handling option M or N. Any invalid signed ASCII digit that does not appear in Table 7-4 is replaced by positive zero ({).

Table 7-4. Valid Replacements for Invalid Signed ASCII Digits

Invalid Signed ASCII Digit	Valid Replacement Digit	Value of Replacement Digit
a, /, 1	A	+1
b, s, S, 2	B	+2
c, t, T, 3	C	+3
d, u, U, 4	D	+4
e, v, V, 5	E	+5
f, w, W, 6	F	+6
g, x, X, 7	G	+7
h, y, Y, 8	H	+8
i, z, Z, 9	I	+9
j	J	-1
k	K	-2
l	L	-3
m	M	-4
n	N	-5
o	O	-6
p	P	-7
q	Q	-8
r	R	-9
-,]	}	0

Using Debug

Debug, the MPE XL System Debugger, allows you to analyze your program's object code while it is executing. With Debug, you can do the following:

- Display data item values.
- Change data item values and continue program execution.
- Set breakpoints at statements.
- Set breakpoints at data items. You can cause the program to break when a specified data item changes value.

Note This section is not a Debug tutorial. It only explains how to obtain run-time addresses that you can use in Debug commands. For information on the Debug commands themselves, refer to the *MPE XL System Debug Reference Manual*.

While using Debug, you need the maps listed in Table 7-5.

Table 7-5. COBOL Maps

Map	Contents of Map	Purpose of Map	How to Get the Map
Symbol table map	Data item offsets.	To display or change data item values and to set breakpoints at data items.	Compile your program with control option MAP.
Verb map	Offsets of program statements from starting addresses in link map.	To set breakpoints at statements.	Compile your program with control option VERBS.
Link map	Actual starting addresses.	To set breakpoints at statements if the program is in chunks, and to find the addresses of OWN or EXTERNAL data.	Link Editor command LISTPROG. The information in the link map is also available from Debug, during program execution, with the PROCLIST command. See the section "Link Map."

This section does the following:

- Explains the symbol table, verb, and link maps.
- Gives an example of a chunked program and its maps.
- Gives an example of nested and concatenated programs and their maps.
- Explains how to get data and program offsets at run time.
- Explains how to find subprogram parameters.
- Explains register meanings.
- Explains how to calculate the addresses of data items.
- Explains how to calculate code addresses.
- Explains how to debug trap errors with Debug.

Symbol Table Map

A symbol table map lists data item offsets and lengths. The offsets and lengths are in bytes, in hexadecimal representation. You need these offsets in order to display or change data item values or set breakpoints at data items.

In a symbol table map, EXTERNAL items have the base EXT. Offsets are offsets from the address of the level 01 EXTERNAL item.

To produce a symbol table map of your program, compile it with the control option MAP (see Chapter 6).

Verb Map

A verb map lists the offsets of the statements in your program. The offsets are in bytes, in hexadecimal representation. They are offsets from the entry point or chunk named in the heading of the verb map page. You need them in order to set breakpoints at statements. To produce a verb map of a COBOL program, compile it with the control option VERBS.

When reading a verb map, remember that some code offsets are offset from the beginning of chunks. See "Maps Example for Chunked Program."

Link Map

A link map lists the following actual starting addresses, which you need in order to set breakpoints at statements and display data. A link map is one way to find the addresses of chunk entry points and OWN and EXTERNAL data. The other way is to use the PROCLIST command when you execute your program under Debug.

- Starting addresses of chunks, if applicable. See "Maps Example for Chunked Program."

  Each chunk name that appears in the verb map is the starting address of a chunk. The code offsets in the verb map are offsets from the starting addresses of chunks. Compiler-generated chunk names are of the form *program_name*$*nnn*$.

- Starting address of main program data storage (relative to DP).

  Other main program data addresses are offsets from this address. You do not need the link map for the main program.

- Starting address of subprogram data storage (one per subprogram).

  For a subprogram compiled with the ANSISUB or SUBPROGRAM option, M$*n*$*program_name* (where *n* is a number) contains the starting address of the subprogram's OWN data. For a subprogram compiled with the DYNAMIC option, SP contains the ending address of the subprogram's data storage (a value that is only available while the subprogram is executing). Other subprogram addresses are offsets from this address.

You can ignore other addresses.

The link map also shows the program entry point. Refer to the Link Editor manual for the commands that produce a link map.

Maps for Chunked Program

For a large program, the executable code is in chunks. Each chunk is in a separate subspace, but they all reside in the same file. When reading a verb map and using Debug, code offsets are offset from the beginning of chunks.

Chunk names are derived from the *program-id*, as explained in Chapter 4. See "Calculating Code Addresses" to determine how to find code addresses of chunks.

Example

The following compiler listing for a chunked program includes a verb map. The following is the compiler command for this example:

```
:COB85XL NC101.PROG;INFO="$CONTROL VERBS"
```

The following is the compiler listing:

```
************************************************************
*      COB85XL - Compile COBOL program.
*                Object file will be $NEWPASS.
************************************************************

PAGE 0001   HP31500A.01.00  [85] (C) HEWLETT-PACKARD CO. 1987


 00001 COBCNTL  001000*    COBCNTL.PUB.SYS  Defaults are:
 00002 COBCNTL  002000*CONTROL LIST,SOURCE,NOCODE,NOCROSSREF,ERRORS=100,
                NOVERBS,WARN
 00003 COBCNTL  003000*CONTROL LINES=60,NOMAP,MIXED,QUOTE=",NOSTDWARN,
                SYNC32,INDEX32
 00004          INFO=$control verbs
 00005 NC1014.1 000100 IDENTIFICATION DIVISION.
 00006 NC1014.1 000200 PROGRAM-ID.
 00007 NC1014.1 000300     NC101.
          ⋮
 02001 NC1014.1 200800 DIV-WRITE-57.
 02002 NC1014.1 200900     MOVE "DIV-TEST-57" TO PAR-NAME.
 02003 NC1014.1 201000     PERFORM PRINT-DETAIL.
 02004 NC1014.1 201100 CCVS-EXIT SECTION.
 02005 NC1014.1 201200 CCVS-999999.
 02006 NC1014.1 201300     GO TO CLOSE-FILES.
```

```
PAGE 0002/COBTEXT  NC101          STATEMENT OFFSETS
                                  Entry = nc101

     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET    STMT   OFFSET ...

      261       3C    317      848    373      F44     428     15CC
      262       40    318      860    374      F44     429     16A8
      263       40    319      860    375      F6C     430     16B4
      264       74    320      890    376      F84     431     16C4
      265       BC    321      8A4    377      F84     432     16CC
      266       D0    322      8B4    378     10C0     433     16CC
      267       E4    323      8E8    379     10F0     434     16E0
      268       EC    324      918    380     10F8     435     16E8
        ⋮
     1401     9470   1444     9988   1487     A0E4    1530     A4E8
     1402     949C   1445     998C   1488     A0EC    1531     A514
     1403     94B4   1446     998C   1489     A0F4
     1404     94B4   1447     99B8   1490     A0F4
     1405     94C4   1448     99D0   1491     A0F4


PAGE 0008/COBTEXT  NC101          STATEMENT OFFSETS
                                  Chunk = nc101$001$

     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET    STMT   OFFSET ...

     1532        8   1587      72C   1642      FEC    1697     1608
     1533        8   1588      738   1643      FEC    1698     1610
     1534       20   1589      73C   1644     1010    1699     1610
     1535       3C   1590      92C   1645     102C    1700     1640
        ⋮
     1978     3C4C   1987     3D98   1996     3E50    2005     3F80
     1979     3C54   1988     3DB0   1997     3E18    2006     3F80
     1980     3C84   1989     3DB0   1998     3F2C
```

Note

The verb map on PAGE 0002 through PAGE 0008 is due to the control option VERBS. Notice the entry name *nc101* on PAGE 0002 and the chunk name *nc101$001$* on PAGE 0008. Offsets are offsets from the addresses under SYM VALUE in the link map.

```
PAGE 0011/COBTEXT  NC101           COBOL ERRORS and WARNINGS


COBOL ERRORS:


LINE #   SEQ #   COL   ERROR   SEV   TEXT OF MESSAGE
------  ------   ---   -----   ---   ---------------


 00957  096400   80     050     W    ARITHMETIC OVERFLOW MAY OCCUR.
 00973  098000   80     050     W    ARITHMETIC OVERFLOW MAY OCCUR.
 00989  099600   80     050     W    ARITHMETIC OVERFLOW MAY OCCUR.
 01005  101200   80     050     W    ARITHMETIC OVERFLOW MAY OCCUR.


0 ERRORS, 0 QUESTIONABLE, 4 WARNINGS

    DATA AREA IS       9FC BYTES.
    CPU TIME = 0:01:20.   WALL TIME = 0:06:19.

END OF PROGRAM
END OF COBOL COMPILATION
```

The following shows a Link Editor session that displays the link map of the chunked program:

```
:LINK $OLDPASS;MAP
HP Link Editor/XL (HP30315A.02.27) Copyright Hewlett-Packard Co 1986

LinkEd> link $OLDPASS;MAP

PROGRAM         : $OLDPASS
XL LIST         :
CAPABILITIES    : BA, IA
NMHEAP SIZE     :
NMSTACK SIZE    :
VERSION         : 85082112


Sym                      C H X P Sym    Sym        Sym     Lset
Name                             Type   Scope      Value   Name
----                     - - - - ----   -----      -----   ----
$START$                  0   3 3 sec_p  univ       00009A74
_start                   0   3 3 sec_p  univ       0000AF68 NC101
nc101                    0   3 3 pri_p  univ       0000AF4C NC101
$RECOVER_END             0       code   univ       0001AF4C
$RECOVER_START           0       code   univ       0001ADD8
$START$                  0       code   univ       00009A90
$UNWIND_END              0       code   univ       0001ADB8
$UNWIND_START            0       code   univ       0001A4F8
_start                   0       code   univ       0000B044 NC101
nc101                    0       code   univ       0000B044 NC101
nc101$001$               0       code   univ       00016094 NC101
M$1                      0       data   local    dp+00000000
.
.
.
```

The entry names `nc101` and `nc101$001$` are identifiable by the type `code` under `SYM TYPE`.

Example Maps for Nested and Concatenated Programs

This compiler listing is for two concatenated programs, MAIN-P and CONCAT-P. The first program, MAIN-P, contains a nested program, NESTED-P. The compiler appends the maps to the end of the listing of all three programs. First the symbol maps and cross reference appear, followed by the verb maps.

Note that the nested program NESTED-P is compiled with $CONTROL DYNAMIC. Because of this, the symbol map shows the address for the data item local to this program to be relative to the stack pointer (SP).

```
PAGE 0001   HP31500A.01.00 [85]  (C) HEWLETT-PACKARD CO. 1987

 00001          INFO=$CONTROL MAP,verbs
 00002          001000 IDENTIFICATION DIVISION.
 00003          001100 PROGRAM-ID. MAIN-P.
 00004          001200 DATA DIVISION.
 00005          001300 WORKING-STORAGE SECTION.
 00006          001400 1 DUMMY-N  GLOBAL         PIC 99  VALUE 88.
 00007          001500 PROCEDURE DIVISION.
 00008          001600 BEGIN.
 00009          001700 CALL "NESTED-P".
 00010          001800
 00011          001810$control dynamic
 00013          001900 IDENTIFICATION DIVISION.
 00014          002000 PROGRAM-ID. NESTED-P.
 00015          002100 DATA DIVISION.
 00016          002200 WORKING-STORAGE SECTION.
 00017          002300 01 LOCAL-DATA  PIC X(15).
 00018          002400 PROCEDURE DIVISION.
 00019          002500 BEGIN.
 00020          002600 MOVE "END IN NESTED-P" TO LOCAL-DATA.
 00021          002700 ADD 1 TO DUMMY-N.
 00022          002800 CALL "CONCAT-P".
 00024          002900 END PROGRAM NESTED-P.
 00025          003000 END PROGRAM MAIN-P.
 00026          003100
 00027          003200 IDENTIFICATION DIVISION.
 00028          003300 PROGRAM-ID. CONCAT-P.
 00029          003400 DATA DIVISION.
 00030          003500 WORKING-STORAGE SECTION.
 00031          003600 01 LOCAL-DATA  PIC X(15).
 00032          003700 PROCEDURE DIVISION.
 00033          003800 BEGIN.
 00034          003900 MOVE "END IN CONCAT-P" TO LOCAL-DATA.
 00036          004000 END PROGRAM CONCAT-P.
```

```
PAGE 0002/COBTEXT  MAIN-P          SYMBOL TABLE MAP
LINE# LVL SOURCE NAME       BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ


WORKING-STORAGE SECTION

00006 01  DUMMY-N          DP+        28         2 DISP      N


PAGE 0003/COBTEXT  MAIN-P          SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME      BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ

STORAGE LAYOUT            (#ENTRYS)

FIRST TIME FLAG, etc.      DP+         8         4
RUN TIME $ . ,             DP+        14         4
SORT/MERGE PLABEL          DP+        18         4
TALLY                      DP+        24         4
USER STORAGE               DP+        24         6


PAGE 0004/COBTEXT  NESTED-P        SYMBOL TABLE MAP
LINE# LVL SOURCE NAME       BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ


WORKING-STORAGE SECTION

00017 01  LOCAL-DATA       SP        -40         F DISP      AN
```

```
PAGE 0005/COBTEXT  NESTED-P        SYMBOL TABLE MAP
LINE# LVL SOURCE NAME       BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ

STORAGE LAYOUT             (#ENTRYS)

FIRST TIME FLAG, etc.       SP        -60         4
RUN TIME $ . ,              SP        -54         4
SORT/MERGE PLABEL           SP        -50         4
TALLY                       SP        -44         4
USER STORAGE                SP        -44        13
Literal pool ~    S$        CODE        0        14


PAGE 0006/COBTEXT  CONCAT-P        SYMBOL TABLE MAP
LINE# LVL SOURCE NAME       BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ


WORKING-STORAGE SECTION

00031 01  LOCAL-DATA       OWN         20         F DISP      AN


PAGE 0007/COBTEXT  CONCAT-P        SYMBOL TABLE MAP
LINE# LVL SOURCE NAME       BASE  OFFSET     SIZE   USAGE     CATEGORY R O J BZ

STORAGE LAYOUT             (#ENTRYS)

FIRST TIME FLAG, etc.       OWN          0         4
RUN TIME $ . ,              OWN          C         4
SORT/MERGE PLABEL           OWN         10         4
TALLY                       OWN         1C         4
USER STORAGE                OWN         1C        13
Literal pool ~    S$        CODE        0        10
```

```
PAGE 0008/COBTEXT  CONCAT-P        STATEMENT OFFSETS
                                    Entry = main_p
     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

        8       1C      9       1C


 PAGE 0009/COBTEXT  CONCAT-P        STATEMENT OFFSETS
                                    Entry = main_p$003$nested_p
     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

       19       60     20       60     21       88     22       E8


 PAGE 0010/COBTEXT  CONCAT-P        STATEMENT OFFSETS
                                    Entry = concat_p
     STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

       33       48     34       48

0 ERRORS, 0 QUESTIONABLE, 0 WARNINGS

    DATA AREA IS       84 BYTES.
    CPU TIME = 0:00:01.  WALL TIME = 0:00:02.
```

Subprogram Parameters

Memory contains two copies of each set of subprogram parameters. The two copies allow multiple entry points.

Parameters are stored in dynamically allocated memory and referenced through the stack pointer (SP, register 30) or the previous stack pointer (PSP).

You can access one copy only if you are at a breakpoint at the entry point of the subprogram. The first four parameters are in registers:

Table 7-6. Registers 23 through 26

Register	Parameter
R26	Parameter one.
R25	Parameter two.
R24	Parameter three.
R23	Parameter four.

The rest of the parameters are stored in memory, beginning with SP-34 and working backward. The fifth parameter is in SP-34, the sixth parameter is in SP-38, and so on. Offsets are hexadecimal. The above SP becomes PSP for data references after the entry point.

The location of the other copy appears in the map. Data is not moved there until the first executable subprogram statement executes. You must access this copy if you are past the subprogram entry point but have not exited the subprogram. All parameters are stored in memory, beginning with PSP-24 and working backward. Each parameter is stored in four bytes.

Register Meanings

The following registers have the following meanings, independent of COBOL:

Table 7-7. Registers 0, 1, and 2

Register	Meaning
R0	Always zero.
R1	Scratch register.
R2	Last return address.

The following registers are always present:

Table 7-8. Registers 27, 30, and 31

Register	Contents
R27	DP, the data pointer.
R30	SP, the stack pointer.
R31	Millicode return address. Millicode is special utility code for COBOL and other languages.

Calculating Addresses of Data Items

The first step in displaying or changing the value of a data item is to find its address:

1. Obtain symbol table and link maps for your program.
2. On the symbol table map, find the base and offset of the data item that you want to display or change. The base is DP, SP, EXT, OWN, or P *nn*, where *nn* is the parameter number.
3. If the base is DP (the address of the DATA DIVISION of the main program), then the address of the data item is DP+*offset*. (DP is different in each XL module.)
4. If the base is SP (the address of the Working Storage of a DYNAMIC subprogram or the temporary cells of the main program), then the address of the data item is SP-*offset*, which is only valid when you are in the subprogram.
5. If the base is EXT (the address of the program's external variables), find the data item in the link map. Add the address that the link map shows for the data item to the offset from the symbol table map. This is the actual address of the data item (DP+*link_addr*+*map_offset*). (DP is different in each XL module.)
6. If the base is OWN (the address of the Working Storage of an ANSISUB or SUBPROGRAM subprogram), then the address of the data item is *address_of_OWN*+*offset*. You must find *address_of_OWN* in the link map. It is the SYM VALUE of M$*n*$ *program_name* (where *n* is a number), which is DP plus a number. (DP is different in each XL module.)
7. If the base is P *nn*, where *nn* is the parameter number (the address of the LINKAGE SECTION), find the address of PARAMETER POINTERS under STORAGE LAYOUT in the symbol table map. It is a negative offset from SP, -*mm*. The address of the first parameter is SP-*mm*. The address of parameter *nn* is:

   `[(SP-`*mm*`) + 4 * `*nn*`] + `*offset*

   (The bracketed value is called the *indirection operator*.)
8. You can obtain data addresses at run time in Debug, using the PROCLIST command (see the *MPE XL System Debug Reference Manual*).

Note The formulas for the first and *nn*th parameters are valid only if you are at or after the first statement of the program. If you are at a breakpoint at the entry point to the subprogram, its parameters have not yet been copied out of the registers.

The following format of the PROCLIST command gives you starting offsets of all data areas associated with all programs in your program file. The global area of the outermost program (main) is shown as M$1, as dp, and as $global$. The COBOL EXTERNAL item EXT-GRADE is downshifted and the hyphen ("-") replaced with an underscore ("_"). EXTERNAL items have named storage associated with them at run time.

The OWN data areas associated with a nested program follow the format M.*nested_program_name*, where each *nested_program_name* follows the format for internal names that is explained in Chapter 3.

Example

```
$3 ($36) nmdebug > proclist,,dataany
571.40000008 $global$
571.40000008 $dp$
571.40000080 ext_grade
571.40000008 M$1
571.40000038 M.main$003$valid_grade
571.40000060 M.main$004$display_bell
```

Note The above display has been edited for conciseness. You can ignore other addresses that appear on your screen.

The above addresses are the values known at link time. The value of dp, 571.40000008, must be subtracted from all values and then added to DP. For example, the address of *ext_grade* is DP+80-8.

For commands that display and change data item values, refer to the *MPE XL System Debug Reference Manual*.

Calculating Code Addresses

You must calculate code addresses in order to set breakpoints in Debug (refer to the *MPE XL System Debug Reference Manual* for breakpoint syntax).

To calculate the code address of a specific statement:

1. In the verb map, find the offset for the statement.
2. If the portion of the verb map that lists the offset for the statement does not have "Chunk = *name*" in its heading, then the statement offset is:

   program_offset + statement_offset

3. If the portion of the verb map that lists the offset for the statement has "Chunk = *name*" in its heading, find the chunk name in the link map. Its SYM VALUE is the chunk address. The statement offset is:

   chunk_offset + statement_offset

To find the offset of a program or chunk, do one of the following:

- Obtain a link map, using the LISTPROG command (see the *Link Editor manual*). Program and chunk names are listed with their offsets.
- Use the PROCLIST command in Debug. This allows you to obtain the offsets at run time. See the *MPE XL System Debug Reference Manual*.

Example

Given a program MAIN that contains one chunk and two nested programs, VALID-GRADE and DISPLAY-BELL, here is one way to get the starting addresses of a main program and any program it contains. Nested program names are qualified with the name of the outermost program that contains them, so if you supply the main program name appended and prepended with wild card characters, and use the parameter *any*, you get the following:

```
$1 ($33) nmdebug > proclist @main@,,any
528.535b     main              [1]
528.5400     main$001$         [2]
528.542b     main.valid_grade  [3]
528.55e3     main.display_bell [4]
```

The first, third, and fourth items are program names and their offsets. The second item is a chunk name and its offset.

Note The above display has been edited for conciseness. You can ignore other addresses that appear on your screen.

Debugging Trap Errors

Debugging trap errors is easier if you have symbol table, verb, and link maps of your program.

The trap error message contains the program counter (pc) address. In the link map, find the value of the pc. The subprogram in which the trap error occurred is the symbol of type *code* with the greatest SYM VALUE not greater than the pc value. Subtract that SYM VALUE from the pc value to get the offset of the trap.

In the verb map for the subprogram in which the trap occurred, find the statement with the greatest offset not greater than the offset of the trap. This is the statement where the trap occurred.

If the subprogram in which the trap occurred is not written in COBOL, the error message may be misleading. However, the pc value, link map, and stack marker trace (which is printed when the program aborts) are accurate.

If you get the system trap error message DATA MEMORY PROTECTION or INVALID ADDRESS, recompile your COBOL program with the control option BOUNDS and rerun it. You will probably get a COBOL trap, because these system traps usually result from values out of bounds.

This section gives at least one debugging example for each type of trap. The types of traps are:

- Illegal ASCII or Decimal Digit.
- Range Error.
- No Size Error.
- PERFORM Stack Overflow.
- Invalid GO TO.
- Address Alignment.
- Intrinsic Function Traps.

This section also explains:

- Trace traps.

Redirecting Output from Traps

The trap examples that follow show output that goes to $STDLIST. The output comes from two sources, the COBOL run-time library and Debug.

Note

The examples that follow print the statement number of the trap as

`Stmt #`*nnn*

However, for optimized programs, the statement number appears as `????`.

This also occurs for statements that generate a large amount of machine instructions. In such cases, see the section "Calculating Code Addresses" in this chapter to determine the statement where the error occurs.

Illegal ASCII Digit

In this example, a program with illegal ASCII digits is executed with the COBRUNTIME value A. The effect of other COBRUNTIME values is explained after the example.

Given the following compiled source program:

```
00001          000100$CONTROL MAP,VALIDATE
00003          001000 IDENTIFICATION DIVISION.
00004          002000 PROGRAM-ID. TRAP-ASCII-DIG.
00005          003000 DATA DIVISION.
00006          004000 WORKING-STORAGE SECTION.
00007          005000 01  BAD-VALUE   PIC X(4) VALUE "12 4".
00008          006000 01  B           REDEFINES BAD-VALUE PIC 9(4).
00009          007000 01  D           PIC 9(4).
00010          008000 PROCEDURE DIVISION.
00011          009000 TRAP-TEST.
00012          009100*****************************************
00013          009200* Trap test with illegal ASCII digit     *
00014          009300*****************************************
00015          010000 DISPLAY "Should produce illegal ASCII digit.".
00016          011000 ADD 1 TO B.
00017          012000 DISPLAY BAD-VALUE.
00018          013000 DISPLAY "Test case did not abort.".
00019          014000 STOP RUN.
```

The section of the map that shows the addresses of items in WORKING-STORAGE is:

```
LINE# LVL SOURCE NAME          BASE  OFFSET     SIZE   USAGE     CATEGORY
WORKING-STORAGE SECTION

00007 01  BAD-VALUE           DP+        30         4 DISP      AN
00008 01  B                   DP+        30         4 DISP      N
00009 01  D                   DP+        34         4 DISP      N
```

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), and the program is compiled with the control option VALIDATE, this program produces the following information when run in session mode. (When run in a job stream, the output may look slightly different. When debugging a program, running it in a session is recommended.)

```
:$OLDPASS
Should produce illegal ASCII digit.
Illegal ASCII digit (COBERR 711)
Program file: $OLDPASS.H60PT2.COBOL74
Trap type = 00000200 (22,00), at pc = 000004D0.000083BF
invalid ascii digit
Source address = 40200030, Source = '12 4'
                   (hex)    Source = '31322034'
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 4d0.0000d1fc print_message$093+$314
$$$$ Trap occurred at: trap_ascii_dig+$74, at Stmt    #16
     PC=4d0.00007ac0 vloop+$c
  0) SP=402210e8 RP=4d0.000083c4 trap_ascii_dig+$7c
* 1) SP=402210e8 RP=4d0.00000000 inx_A0000+$14
     (end of NM stack)
$$$$ The address $40200030 = DP+$28 may be in main, SUBPROGRAM or EXTERNAL
```

The most important pieces of information are flagged by the string "$$$$." The first gives you the number of the statement that preceded the trap. The second gives you the address of the data item that caused the trap. If you look up DP+$28 in the map, you find that the data item BAD-VALUE caused this trap. (Since the output above says that the data item that caused the trap could be in a main program, be in a subprogram, or be an EXTERNAL item, you must study the maps and data of your program to determine that BAD-VALUE is the culprit. The statement number (#16) may also help determine the problem.)

Beneath the line flagged "$$$$" above is a stack trace that shows where in your code the program aborted. The highest name on this stack that is the name of one of your programs has the exact address in the program file where the abort occurred. This may be useful if you use Debug. Refer to the *MPE XL System Debug Reference Manual* for more information.

If you change the value of COBRUNTIME and rerun the above program, program behavior changes as described below.

COBRUNTIME Value	Change in Program Behavior
I	The trap occurs, but the program ignores it and continues.
M	The trap occurs, the above trap information is printed, the illegal digit is replaced by a legal digit, and the program continues.
N	The trap occurs, the illegal digit is replaced by a legal digit, and the program continues.
C	The trap occurs, the above trap information is printed, and the program continues.

The following example shows an illegal ASCII digit error with an invalid sign. The error occurs when attempting to display variables N4 and N4R.

```
C
PAGE 0001         COBOL II/XL   HP31500A.04.03  [85]    TUE, JUL  9, 1991,  5:18
                 PM     Copyright Hewlett-Packard CO. 1987

 00001          001000$CONTROL VALIDATE
 00002          001100 IDENTIFICATION DIVISION.
 00003          001200 PROGRAM-ID.
 00004          001300     COBMAIN.
 00005          001400 ENVIRONMENT DIVISION.
 00006          001500 DATA DIVISION.
 00007          001600 WORKING-STORAGE SECTION.
 00008          001700 01  A1     PIC X VALUE SPACES.
 00009          001800 01  N1     PIC 99 VALUE 0.
 00010          001900 01  N2     PIC 99 VALUE 0.
 00011          002000 01  N3     PIC S99 VALUE 0.
 00012          002010 01  N4     PIC 999.
 00013          002020 01  N4R REDEFINES N4 PIC S999.
 00014          002100 PROCEDURE DIVISION.
 00015          002200 FIRST-PARA.
 00016          002300    MOVE 10 TO N1.
 00017          002400    MOVE N1 TO N3.
 00018          002500    DISPLAY N1, N3.
 00019          002510    MOVE 111 TO N4.
 00020          002520    DISPLAY N4, N4R.
 00021          002600    STOP RUN.

0 ERROR(s), 0 QUESTIONABLE, 0 WARNING(s)

    DATA AREA IS       44 BYTES.
    CPU TIME = 0:00:01.  WALL TIME = 0:00:01.
```

```
10+10
Illegal ASCII digit (COBERR 711)
Program file: $OLDPASS.USER2.COBOL74
Trap type = 00000200 (22,01), at pc = 000000B9.00005E07
invalid ascii digit     (S->S)
Source address = 4033A0BC, Source = '111'
                 (hex)    Source = '313131'
DEBUG/XL A.47.01

HPDEBUG Intrinsic at: 678.00351270 cob_trap.print_message+$5c4
$$$$ Trap occurred at: cobmain+$1b4, at Stmt    #20
     PC=b9.00005e04 cobmain+$1b4
* 0) SP=4033a0f0 RP=b9.00000000 $$inx_A0000+$14
     (end of NM stack)
$$$$ The address $4033a0bc = SP-$34 is in cobmain (TEMPCELL or $DYNAMIC)
============================================================

**** COB_QUIT 711 ****

ABORT: $OLDPASS.USER2.COBOL74
NM SYS   a.00944414 dbg_abort_trace+$2c
NM USER  678.003502f8 COB_QUIT+$b8
NM SYS   a.004403d0 user_trap_caller+$e4
 --- Interrupt Marker
NM PROG  b9.00005e04 cobmain+$1b4
```

Range Error

These errors may occur when a subscript or index references an array out of bounds.

Given the following compiled source program:

```
00001          001000$CONTROL BOUNDS,MAP
00003          001100 IDENTIFICATION DIVISION.
00004          001200 PROGRAM-ID. BOUNDSEXAMPLE.
00005          001300* This program has a subscript out of bounds.
00006          001400 DATA DIVISION.
00007          001500 WORKING-STORAGE SECTION.
00008          001600 01.
00009          001700   05                      PIC X.
00010          001800   05 XX                   PIC X.
00011          001900 01.
00012          002000   05 Y OCCURS 80 TIMES    PIC X.
00013          002100 01 Z                      PIC S9(9) COMP VALUE 0.
00014          002200 PROCEDURE DIVISION.
00015          002300 P1.
00016          002400************************************************
00017          002500* Trap test with range error                   *
00018          002600************************************************
10019          002700     MOVE -5 TO Z.
00020          002800     MOVE  Y(Z) TO XX.
00021          002900     DISPLAY "Test case did not abort.".
00022          003000     STOP RUN.
```

The section of the map that shows the addresses of items in WORKING-STORAGE is:

```
LINE# LVL SOURCE NAME          BASE  OFFSET     SIZE   USAGE     CATEGORY

WORKING-STORAGE SECTION

00008 01  FILLER              DP+        28         2 DISP      AN
00009 05   FILLER             DP+        28         1 DISP      AN
00010 05   XX                 DP+        29         1 DISP      AN
00011 01  FILLER              DP+        2C        50 DISP      AN
00012 05   Y                  DP+        2C         1 DISP      AN
00013 01  Z                   DP+        7C         4 COMP      NS
```

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), and the program is compiled with the control option BOUNDS, this program produces the following information when run in a session. (When run in a job stream, the output may look slightly different. When debugging a program, running it in a session is recommended.)

```
:$OLDPASS
SUBSCRIPT/INDEX/REFMOD/DEP-ON out of BOUNDS (COBERR 751)
Program file: $OLDPASS.H60PT2.COBOL74
Trap type = 00080000 (12,00), at pc = 00000551.000077CB
range error
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 551.0000c594 print_message$093+$314
$$$$ Trap occurred at: boundsexample+$30, at Stmt    #20
    PC=551.000077c8 boundsexample+$30
* 0) SP=402210d8 RP=551.00000000
    (end of NM stack)
$$$$ The variable -5 <   1 (limit)
```

The most important pieces of information are flagged by the string "$$$$." The first gives you the number of the statement that preceded the trap. The second gives you the bad value of the index or subscript, compared to the limit.

Beneath the line flagged "$$$$" above is a stack trace that shows where in your code the program aborted. The highest name on this stack that is the name of one of your programs has the exact address in the program file where the abort occurred. This may be useful if you use Debug. Refer to the *MPE XL System Debug Reference Manual* for more information.

No Size Error

The No Size Error trap sometimes occurs where it is not possible to specify ON SIZE ERROR, as in an expression in a relational condition. If the same expression were in a COMPUTE statement, the ON SIZE ERROR phrase would execute.

In the following compiled source program, division by zero causes the ON SIZE ERROR trap:

```
0001          001000 identification division.
0002          001100 program-id. zerotrap.
0003          001200 data division.
0004          001300 working-storage section.
0005          001400 77  n                    pic 999.
0006          001500 77  bad-value           pic s999 value zero.
0007          001600 procedure division.
0008          001700 house.
0009          001800*******************************************
0010          001900* Divide by Zero with no Size Error Phrase. *
0011          002000*******************************************
0012          002100     Display "Should give NO SIZE ERROR PHRASE".
0013          002200     Compute n = 1 / BAD-VALUE.
0014          002300     Display "Test case did not abort".
0015          002400     Stop run.
```

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), this program produces the following information when run in a session. (When run in a job stream, the output may look slightly different. When debugging a program, running it in a session is recommended.)

```
:$OLDPASS
Should give NO SIZE ERROR PHRASE
No SIZE ERROR phrase (COBERR 747)
Program file: $OLDPASS.H60PT2.COBOL74
Trap type = 00000002 (30,00), at pc = 00000531.0000835B
integer divide by 0
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 531.0000d124 print_message$093+$314
$$$$ Trap occurred at: zerotrap+$a0, at Stmt    #13
    PC=531.00006774 small_divisor+$8
  0) SP=402210f0 RP=531.00008360 zerotrap+$a8
* 1) SP=402210f0 RP=531.00000000 inx_A0000+$14
    (end of NM stack)
```

The most important piece of information is flagged by the string "$$$$," which tells you the number of the statement that preceded the trap.

Beneath the line flagged "$$$$" above is a stack trace that shows where in your code the program aborted. The highest name on this stack that is the name of one of your programs has the exact address in the program file where the abort occurred. This can be useful if you use Debug. Refer to the *MPE XL System Debug Reference Manual* for more information.

PERFORM Stack Overflow

Each program has a PERFORM stack, used to keep track of which paragraphs are executing and where to return after execution of a paragraph. This stack can overflow if your program has control-flow bugs, as the following example does. It has recursive PERFORM statements.

Given the following compiled source program:

```
00001          001000$CONTROL BOUNDS,VERBS
00003          001100 IDENTIFICATION DIVISION.
00004          001200 PROGRAM-ID. PERFORM-TRAP.
00005          001300 PROCEDURE DIVISION.
00006          001400 TRAP-TEST.
00007          001500 DISPLAY "Should give paragraph stack overflow.".
00008          001600****************************************
00009          001700* Trap test with recursive performs.     *
00010          001800****************************************
00011          001900    PERFORM PAR-A THROUGH PAR-B.
00012          002000    GO TO COMMON-EXIT.
00013          002100 PAR-A.
00014          002200    PERFORM PAR-B.
00015          002300 PAR-B.
00016          002400    PERFORM PAR-A.
00017          002500 COMMON-EXIT.
00018          002600    DISPLAY "Test case did not abort.".
00019          002700    STOP RUN.
```

Here is the verb map that shows the code offsets of the program statements:

```
PAGE 0002/COBTEXT  PERFORM-TRAP    STATEMENT OFFSETS
                                  Entry = perform_trap

    STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   STMT   OFFSET   ...

       6       3C     12       80     15       B4     18       E0
       7       3C     13       88     16       B4     19      110
      11       68     14       88     17       E0
```

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), and the program is compiled with the control option BOUNDS, this program produces the following information when run in a session. (When run in a job stream, the output may look slightly different. When debugging a program, running it in a session is recommended.)

```
:$OLDPASS
Should give paragraph stack overflow.
Paragraph stack overflow (COBERR 748)
Program file: $OLDPASS.H60PT2.COBOL74
Trap type = 00800000 (08,00), at pc = 00000531.00007873
paragraph stack overflow
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 531.0000c64c print_message$093+$314
$$$$ Trap occurred at: perform_trap+$b8, at Stmt    #16
    PC=531.00007870 perform_trap+$b8
* 0) SP=40221278 RP=531.00000000
    (end of NM stack)
Perform stack for COBOL program: perform_trap+$b8
Return at end of procedure     #2 to perform_trap+$a0     Stmt    #14
Return at end of procedure     #1 to perform_trap+$cc     Stmt    #16
Return at end of procedure     #2 to perform_trap+$a0     Stmt    #14
Return at end of procedure     #1 to perform_trap+$cc     Stmt    #16
Return at end of procedure     #2 to perform_trap+$a0     Stmt    #14
Return at end of procedure     #1 to perform_trap+$cc     Stmt    #16
   ⋮
Return at end of procedure     #1 to perform_trap+$cc     Stmt    #16
Return at end of procedure     #2 to perform_trap+$a0     Stmt    #14
Return at end of procedure     #2 to perform_trap+$80     Stmt    #11
End of perform stack
```

The statement number preceding the trap is flagged by the string "$$$$" and the contents of the PERFORM stack is printed. The phrase "Return at end of procedure #*num*" means that the paragraph *num* has been called, where paragraph zero is the first paragraph in the source program, paragraph one is the second, and so on. (When sections are called, the number of the last paragraph in the section appears.) This PERFORM stack trace also tells you from where the paragraph was called: that is the statement number on the far right.

Beneath the line flagged "$$$$" above is a stack trace that shows where in your code the program aborted. The highest name on this stack that is the name of one of your programs has the exact address in the program file where the abort occurred. This can be useful if you use Debug. Refer to the *MPE XL System Debug Reference Manual* for more information.

Invalid GO TO

This example illustrates an invalid GO TO trap, which occurs at an unaltered GO TO statement:

```
00001          001000$CONTROL BOUNDS
00002          001100 IDENTIFICATION DIVISION.
00003          001200 PROGRAM-ID. TRAPGOTO.
00004          001300 PROCEDURE DIVISION.
00005          001400 TRAP-TEST.
00006          001500*****************************************
00007          001600* Trap test with Invalid Goto            *
00008          001700*****************************************
00009          001800    DISPLAY "Should produce illegal goto trap.".
00010          001900 PAR.
00011          002000    GO TO.
00012          002100 PAR2.
00013          002200    ALTER PAR TO PAR2.
00014          002300 COMMON-EXIT.
00015          002400    DISPLAY "Test case did not abort.".
00016          002500    STOP RUN.
```

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), and the program is compiled with the control option BOUNDS, this program produces the following information when run in a session. (When run in a job stream, the output may look slightly different. When debugging a program, running it in a session is recommended.)

```
:$OLDPASS
Should produce illegal goto trap.
Invalid GOTO (COBERR 754)
Program file: $OLDPASS.H60PT2.COBOL74
Trap type = 00100000 (11,00), at pc = 00000551.00007817
nil pointer reference
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 551.0000c5ec print_message$093+$314
$$$$ Trap occurred at: trapgoto+$54, at Stmt    #11
    PC=551.00007814 trapgoto+$54
* 0) SP=402210d8 RP=551.00000000
    (end of NM stack)
```

The most important piece of information is flagged by the string "$$$$." It tells you the statement number of the unaltered GO TO statement.

Beneath the line flagged "$$$$" above is a stack trace that shows where in your code the program aborted. The highest name on this stack that is the name of one of your programs has the exact address in the program file where the abort occurred. This can be useful if you use Debug. Refer to the *MPE XL System Debug Reference Manual* for more information.

Address Alignment

Address alignment traps are caught by the Link Editor. You must link with PARMCHECK=0 AND purposely ignore the Link Editor warnings to cause one of these traps.

This example illustrates parameter misalignment, where a parameter is passed to a COBOL subprogram that is not 32-bit aligned:

```
00001          001000$CONTROL MAP
00002          001100 IDENTIFICATION DIVISION.
00003          001200 PROGRAM-ID. TRAPMAIN.
00004          001300 DATA DIVISION.
00005          001400 WORKING-STORAGE SECTION.
00006          001500 01.
00007          001600   05                   PIC X.
00008          001700   05 XX                PIC X.
00009          001800 PROCEDURE DIVISION.
00010          001900 P1.
00011          002000*****************************************
00012          002100* Trap tests with alignment              *
00013          002200*****************************************
00014          002300 DISPLAY "Should produce alignment trap.".
00015          002400 CALL "Trapsub" USING XX.
00016          002500 DISPLAY "Test case did not abort.".
00017          002600 STOP RUN.
```

This subsection of the map shows the addresses of items in WORKING-STORAGE. Notice that parameter XX is on an odd byte boundary (DP+29).

```
LINE# LVL SOURCE NAME          BASE  OFFSET     SIZE   USAGE     CATEGORY

WORKING-STORAGE SECTION

00006 01  FILLER              DP+        28         2 DISP      AN

00007 05   FILLER             DP+        28         1 DISP      AN
00008 05   XX                 DP+        29         1 DISP      AN
```

COBOL subprograms normally assume worst case (byte) alignment, which should never trap. To cause an address alignment trap, you must do one more thing: compile the subprogram with OPTFEATURES = LINKALIGNED (this option generates faster code, which assumes parameters are 32-bit-aligned).

Here is the compiled subprogram, called by the above main program:

```
00001          001000$CONTROL MAP,SUBPROGRAM,BOUNDS
00003          001100$CONTROL OPTFEATURES=LINKALIGNED
00003          001200 IDENTIFICATION DIVISION.
00004          001300 PROGRAM-ID. TRAPSUB.
00005          001400 DATA DIVISION.
00006          001500 LINKAGE SECTION.
00007          001600 01 XX          PIC X.
00008          001700 PROCEDURE DIVISION USING XX.
00009          001800 P1.
00010          001900    MOVE "**" TO XX.
00011          002000    EXIT PROGRAM.

LINE# LVL SOURCE NAME          BASE  OFFSET     SIZE   USAGE     CATEGORY
LINKAGE SECTION

00007 01  XX                   P 00        0         1 DISP      AN
```

Here is the link command, with parameter checking turned off (if parameter checking is not turned off, the link fails to produce a program file):

```
:link from=strapm,straps;to=ptrapm;PARMCHECK=0
HP Link Editor/XL (HP30315A.00.23) Copyright Hewlett-Packard Co 1986

LinkEd> link from=strapm,straps;to=ptrapm;PARMCHECK=0
INCOMPATIBLE ALIGNMENT: trapsub (STRAPM, STRAPS)  (LINKWARN 1504)
(PARAMETER #1)
```

The Link Editor tells you which program is being called with the bad parameter.

When COBRUNTIME is not set (that is, when it has its default value) or when COBRUNTIME is set to A (Abort), this program produces the following information when run in a session. (When run in a job stream, the output may look slightly different When debugging a program, running it in a session is recommended.)

```
:ptrapm
SHOULD PRODUCE ALIGNMENT TRAP (753)
Address alignment error (COBERR 753)
Program file: PTRAPM.H60PT2.COBOL74
Trap type = 00200000 (10,00), at pc = 000004D0.0000790F
bad address alignment
DEBUG/XL A.01.00

HPDEBUG Intrinsic at: 4d0.0000c724 print_message$093+$314
$$$$ Trap occurred at: trapsub+$4c, at ????
    PC=4d0.0000790c trapsub+$4c
* 0) SP=402212b8 RP=4d0.00007814 trapmain+$54
  1) SP=402210d8 RP=4d0.00000000
    (end of NM stack)
$$$$ Called from: trapmain+$54, at Stmt    #15
$$$$ The address $40200031 = DP+$29 may be in main, SUBPROGRAM or EXTERNAL
```

The lines with "$$$$" in the above example indicate: the program where the abort occurred (*trapsub*), the address in the program with the CALL statement (*trapmain*), and the address (DP+$29) of the data item in question.

Invalid Decimal Data in NUMERIC Class Condition

This example shows how you can force the NUMERIC class condition on packed decimal data to be true in certain cases where it would otherwise be false. By specifying I in column 9 of COBRUNTIME, any NUMERIC class condition on packed decimal data is true in the following cases:

- A signed value in an unsigned PACKED-DECIMAL field.
- An unsigned value in a signed PACKED-DECIMAL field.
- Any invalid sign nibble.

This matches the behavior of HP COBOL II/V programs. Unless you put I in column 9 of COBRUNTIME, the above conditions cause any NUMERIC class condition to be false.

The following is an example program that contains invalid packed decimal data:

```
WORKING-STORAGE SECTION.
01  DECIMAL-NO-SIGN   PIC 9(3) USAGE PACKED-DECIMAL VALUE 123.
01  DECIMAL-SIGN    REDEFINES DECIMAL-NO-SIGN
                      PIC S9(3) USAGE PACKED-DECIMAL.
PROCEDURE DIVISION.
PARA-001.
    DISPLAY "DECIMAL-NO-SIGN is ", DECIMAL-NO-SIGN.
    DISPLAY "DECIMAL-SIGN is ", DECIMAL-SIGN.
    IF DECIMAL-NO-SIGN IS NUMERIC
        DISPLAY "DECIMAL-NO-SIGN is NUMERIC."
    ELSE DISPLAY "DECIMAL-NO-SIGN is not NUMERIC."
    END-IF.
    IF DECIMAL-SIGN IS NUMERIC
        DISPLAY "DECIMAL-SIGN is NUMERIC."
    ELSE DISPLAY "DECIMAL-SIGN is not NUMERIC."
    END-IF.
    STOP RUN.
```

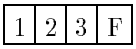
When this program is compiled and run in Native Mode with column 9 of COBRUNTIME set to anything but I, the program displays the following:

```
DECIMAL-NO-SIGN is 123
DECIMAL-SIGN is +123
DECIMAL-NO-SIGN is NUMERIC.
DECIMAL-SIGN is not NUMERIC.    DECIMAL-SIGN contains an invalid sign.
```

When this program is compiled and run either in Native Mode with column 9 of COBRUNTIME set to I, or in Compatibility Mode, it displays the following:

```
DECIMAL-NO-SIGN is 123
DECIMAL-SIGN is +123
DECIMAL-NO-SIGN is NUMERIC.
DECIMAL-SIGN is NUMERIC.        Invalid sign still makes NUMERIC test true.
```

The following diagram shows the contents of the single data item named by DECIMAL-NO-SIGN and redefined by DECIMAL-SIGN. The values shown are hexadecimal. Each box in the diagram represents one packed-decimal digit, which is 4 bits:



The rightmost position is the sign of the data value. The F in this position means the value is unsigned. Any valid signed PACKED-DECIMAL data has either the hexadecimal value C for a positive value, or D for a negative value, in the rightmost 4 bits. For more information, see the section "USAGE IS PACKED-DECIMAL or COMPUTATIONAL-3" in Chapter 7 of the *HP COBOL II/XL Reference Manual*.

Traps with COBOL Functions

In the following example, the parameter for the FACTORIAL function is out of range.

```
00001          000100$control post85
00001          001000 identification division.
00002          001100 program-id. trap-fact.
00003          001200 data division.
00004          001300 working-storage section.
00005          001400 01  n                    pic s9(18) comp.
00006          001500 01  bad-value            pic s999 comp value -1.
00007          001600 procedure division.
00008          001700 p1.
00009          001800*********************************************
00010          001900* factorial(-1) with no size error phrase.  *
00011          002000*********************************************
00012          002100     display "should give no size error phrase"
00013          002200     compute n = function factorial(bad-value)
00014          002300     display "test case did not abort".
```

When run, the following is output:

```
should give no size error phrase
No SIZE ERROR phrase (COBERR 747)
Program file: $OLDPASS.PUBS.COBOL74
Trap type = 80000000 (00,00), at pc = 0000035D.000053C7
assertion trap
DEBUG/XL A.47.01

HPDEBUG Intrinsic at: 118.00351270 cob_trap.print_message+$5c4
$$$$ Trap occurred at: trap_fact+$6c, at Stmt    #13
     PC=35d.000053c4 trap_fact+$6c
* 0) SP=4033a0c8 RP=35d.00000000
     (end of NM stack)
============================================================
```

In the next example, for a function that uses a procedure in the Compiler Library, an invalid parameter is detected by the Compiler Library routine. The Compiler Library routine generates the math library error message and the COBOL trap handler provides the statement number of the source line that produced the trap.

```
00001          000100$control post85
00001          001000 identification division.
00002          001100 program-id. trap-sqrt.
00003          001200 data division.
00004          001300 working-storage section.
00005          001400 01  n                    pic s999v999 comp.
00006          001500 01  bad-value            pic s999 comp value -1.
00007          001600 procedure division.
00008          001700 p1.
00009          001800*******************************************
00010          001900* sqrt(-1) with no size error phrase.       *
00011          002000*******************************************
00012          002100     display "should give no size error phrase"
00013          002200     compute n rounded = function sqrt(bad-value)
00014          002300     display "test case did not abort".
```

When run, the following is output:

```
should give no size error phrase
No SIZE ERROR phrase (COBERR 747)
Program file: $OLDPASS.PUBS.COBOL74
Library trap type = 0000000B (11)
DEBUG/XL A.47.01

HPDEBUG Intrinsic at: 7a0.00016ec4 cob_trap_lib.print_message+$158
$$$$ Call occurred at: trap_sqrt+$68, at Stmt    #13
     PC=7a0.00016ec4 cob_trap_lib.print_message+$158
  0) SP=4033a480 RP=7a0.00017014 COB_TRAP_LIB+$a8
  1) SP=4033a428 RP=7a0.0000d504 COB_TRAP_LIB+$8
       export stub: 109.00377d14 FTN_GETLIBTRAP+$6c
  2) SP=4033a250 RP=109.00377c74 ?FTN_GETLIBTRAP+$8
       export stub: 109.005b1ddc DINVALIDERR+$e4
  3) SP=4033a1d8 RP=109.005b12bc _dsqrterr+$1c
  4) SP=4033a158 RP=109.005b0c98 label2+$10
  5) SP=4033a128 RP=109.005b0b84 ?FTN_DSQRT$+$8
       export stub: 308.00005398 trap_sqrt+$68
* 6) SP=4033a0d0 RP=308.00000000
     (end of NM stack)
============================================================
*** MATH LIBRARY ERROR 28: DSQRT(X): X < 0.0 OR X = NaN
```

The next example shows an IEEE floating point error.

```
00001          001000$control post85
00001          001100 identification division.
00002          001200 program-id. trap-float-zero.
00003          001300 data division.
00004          001400 working-storage section.
00005          001500 01  n                    pic s999v999 comp.
00006          001600 01  bad-value            pic s999 comp value zero.
00007          001700 procedure division.
00008          001800 p1.
00009          001900*******************************************
00010          002000* divide by zero with no size error phrase. *
00011          002100*******************************************
00012          002200     display "should give no size error phrase".
00013          002300     compute n rounded = 1 / function sin(bad-value)
00014          002400     display "test case did not abort".
```

When run, the following is output:

```
should give no size error phrase
No SIZE ERROR phrase (COBERR 747)
Program file: $OLDPASS.PUBS.COBOL74
Trap type = 00020000 (14,27), at pc = 0000035D.000053B7
ieee divide by 0        DIV
        operand 1 = 3FF0000000000000
        operand 2 = 0000000000000000
        result    = 47C3500000000000
DEBUG/XL A.47.01

HPDEBUG Intrinsic at: 118.00351270 cob_trap.print_message+$5c4
$$$$ Trap occurred at: trap_float_zero+$7c, at Stmt    #13
     PC=35d.000053b4 trap_float_zero+$7c
* 0) SP=4033a0d0 RP=35d.00000000
     (end of NM stack)
============================================================
```

Trace Traps

Trace traps are "global" breakpoints that are easy to set up and use, providing a very useful tool for quickly isolating program problems within Debug.

Trace traps are global in that they apply to all paragraphs and sections, entry and/or exit points. The COBOL compiler generates them within an object program, and once generated, they can be armed and used with Debug on MPE XL.

To use trace traps with your program:

1. Compile your program with the control option SYMDEBUG. This causes the compiler to generate trace trap breakpoints in your code. (Note that these breakpoints are not available if the SYMDEBUG=XDB option is specified.)
2. Arm the trace trap breakpoints with the Debug command TRAP (see the syntax that follows).

Syntax. Type the underlined part in response to the Debug prompt.

```
$nmdebug> TRAP [trap_name]   [option]
```

Parameters.

trap_name	One of the following:	
	`BEGIN_PROCEDURE`	Stops the program at the entry to each procedure. For a COBOL program, "procedure" means program.
	`END_PROCEDURE`	Stops the program at the exit from each procedure. For a COBOL program, "procedure" means program.
	`LABELS`	Stops the program at every section and paragraph.
	`EXIT_PROGRAM`	Stops the program at program exit point. For a COBOL program, this is usually the STOP RUN statement.
	`ENTER_PROGRAM`	Stops the program at the program entry point.
	`TRACE_ALL`	All of the above.
option	One of the following:	
	`LIST`	List the current setting of *trap_name*. This is the default.
	`ARM`	Arm the trap specified by *trap_name*.
	`DISARM`	Disarm the trap specified by *trap_name*.

Example. If you type the underlined command in response to the Debug prompt, the program stops at every section and paragraph.

```
$nmdebug> TRAP LABELS ARM
```

Note The SYMDEBUG=TOOLSET control option significantly increases the amount of object code and execution time for your program, so use it only while you are debugging your program. When the program works, recompile it without SYMDEBUG.

Symbolic Debuggers

The symbolic debuggers HP Symbolic Debugger/XL and HP TOOLSET/XL are more powerful and easier to use than Debug, the MPE XL System Debugger. Maps are unnecessary, commands are simple, and labeled function keys are available. If these debuggers are available on your system, you can use them to do the following:

- Reference data items by their names, instead of their addresses.
- Display data item values.
- Change data item values.
- Set and clear breakpoints, with optional frequency and proceed counts.
- Edit the source file at a breakpoint (HP TOOLSET/XL only).
- Display a compiler listing at a breakpoint.
- Trace and retrace program flow.
- Trace changes in data item values between paragraphs.

HP Symbolic Debugger/XL

HP Symbolic Debugger/XL is a powerful, full-featured symbolic debugger that is interactive at the source level. This allows you to examine the program state in which an error occurs, correct the situation, and resume execution or abort the program.

For detailed information about HP Symbolic Debugger/XL and its use, refer to the *HP Symbolic Debugger/XL Reference Manual*.

HP TOOLSET/XL

HP TOOLSET/XL provides an integrated programming environment, including a symbolic debugger, a workspace file manager with version control, function keys set to compile and prepare source files, and a full-screen editor.

For detailed information about HP TOOLSET/XL and its use, refer to the *HP TOOLSET/XL Reference Manual*.

Compiler Limits

The following compiler limits are maximum values that you cannot change. If your program exceeds them, the compiler prints internal error messages. You can work around some of these limits, as noted.

Table 7-9. Compiler Limits

Limited Entity	Maximum Value	Work-Around
Value pairs per *condition-name*. (A value pair is a *literal*, such as 3, or of the form *literal* THRU *literal*, such as *10 THRU 20*.)	500	Describe the condition with more than one *condition-name* and use AND or OR.
GO TO DEPENDING ON paragraph labels.	500	Split the GO TO DEPENDING ON statement into two statements.
Size and number of macros.	Program dependent. Compiler prints message 461 (dynamic error, out of space) when it reaches the limit of 2000.	Eliminate unused macros. (The compiler option $CONTROL CROSSREF indicates where macros are used.) Break up macros into smaller macros. Or, instead of using macros, use COPY REPLACING.
Total number of *pseudo-text* lines in all COPY and REPLACE statements.	Total Approximately 60.	None.
Symbol table entries.	Depends on number and lengths of data item names. If 80% or more of the symbol table and intermediate data structure is used, the compiler prints the percentage. This indicates that the compiler is reaching its limit, you should consider breaking your program into smaller pieces.	None, once the symbol table is full.
Data area.	2^{30} bytes.	None.
Paragraph size.	100K bytes.	Break paragraph into smaller paragraphs.
Sort files.	50	Instead of performing one sort, perform two sorts and merge the sorted files into a single file.
Operands per INSPECT, STRING, or UNSTRING statement.	Approximately 500 operands (fewer if they are subscripted, reference modified, or literals).	Break up statements.

Table 7-9. Compiler Limits (continued)

Limited Entity	Maximum Value	Work-Around
Nonredefined 01 and 77 data items in the LINKAGE SECTION.	255	Put separate data items into records in the subprogram and the calling program.
PICTURE string.	30 characters.	None.
Call BY CONTENT.	64K bytes per call.	Pass extra parameters by reference.
Qualifiers per qualified data name.	50	None.
Identifier.	30 characters (ANSI Standard).	None. (Use a shorter name).
Numeric PICTURE.	18 digits (31 in the intermediate result).	None.
Levels of a PERFORM ... VARYING statement.	7	Use nested PERFORM statements.
DISPLAY operands.	Approximately 500 operands (fewer if they are subscripted, reference modified, or numeric literals).	Use DISPLAY with no advancing.
Depth to which you can nest parentheses in an expression (where a single set of parentheses represents a depth of one).	30	Break the expression into several statements.
Nonnumeric literal.	255 characters.	None.
TOP of LINAGE clause.	63	None.
Nested IF statements.	30	Use EVALUATE or binary IF statements.
Nested and concatenated programs in the same file.	999	Put some programs in separate files.
External data names and files.	4000	Combine items into records.
Corresponding pairs in ADD, SUBTRACT, MOVE, or CORR.	500	Break up groups.
Perform table.	MAX(N+10,50) where N is the total different perform exits.	Your program is not ANSI standard.
Maximum amount of tempcells used for functions.	64K bytes.	Break up statements or use reference modification to break up function parameters.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X