
HP 3000 Computer Systems

**HP FORTRAN 77/iX
Programmer's Guide**



HP Part No. 31501-90011
Printed in U.S.A. June 1992

E0692
Fourth Edition

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 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.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright © 1988, 1989, 1990, 1992 by Hewlett-Packard Company

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	December 1988	31501A.02.00
Second Edition	October 1989	31501A.03.05
Third Edition	December 1990	31501A.04.11
Fourth Edition	June 1992	31501A.04.31

Preface

This is the reference manual for the HP FORTRAN 77 programming language as it is implemented on the MPE/iX operating system. This manual assumes that the reader has been trained in the FORTRAN language and knows FORTRAN programming techniques.

Chapter Summary

This manual is organized into the following chapters:

chapter 1	Describes the factors that influence storage allocation for a FORTRAN variable.
chapter 2	Describes how to use formatted and list-directed input/output statements.
chapter 3	Describes input/output operations with disk and internal files.
chapter 4	Describes some FORTRAN 77 file operations under the operating system.
chapter 5	Describes subroutines, functions, and block data subprograms.
chapter 6	Describes methods that make programs more efficient. Discusses the optimizer.
chapter 7	Describes how to port FORTRAN 77 programs from other systems onto HP systems and how to make new programs easily transportable between HP systems. Discusses portability topics related to the operating system.
chapter 8	Discusses the interface between FORTRAN and other languages.
chapter 9	Discusses facilities in FORTRAN 77 that are useful in debugging programs.

Additional Documentation

More information on HP FORTRAN 77 and related topics can be found in the following manuals:

- *HP FORTRAN 77/iX Reference (31501-90010)*

This manual is a complete reference of all HP FORTRAN 77/iX features.

- *HP FORTRAN 77/iX Migration Guide (31501-90004)*

This manual contains information on how to run FORTRAN 66/V and HP FORTRAN 77/V programs on the MPE/iX operating system and how to convert them to HP FORTRAN 77/iX programs.

In addition, the following manuals are referenced in this manual:

- *MPE/iX Intrinsic Reference Manual (32650-90028)*
- *HP Symbolic Debugger/iX User's Guide (31508-90003)*
- *HP Toolset/iX Reference Manual (36044-90001)*

Conventions

UPPERCASE

In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

`COMMAND`

can be entered as any of the following:

`command` `Command` `COMMAND`

It cannot, however, be entered as:

`comm` `com_mand` `comamnd`

italics

In a syntax statement or an example, a word in italics represents an optional parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

`COMMAND filename`

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

`(filename):(filename)`

underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

Do you want to continue? >> yes

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either **ON** or **OFF**:

$$\text{COMMAND } \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \end{array} \right\}$$

[]

In a syntax statement, brackets enclose optional elements. In the following example, **OPTION** can be omitted:

$$\text{COMMAND } \textit{filename} \text{ [OPTION]}$$

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select **OPTION** or *parameter* or neither. The elements cannot be repeated.

$$\text{COMMAND } \textit{filename} \left[\begin{array}{l} \text{OPTION} \\ \textit{parameter} \end{array} \right]$$

[...]

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

$$[, \textit{parameter}] [\dots]$$

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

$$[\textit{parameter}] [, \dots]$$

Conventions (continued)

| ... |

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **A**, **AB**, **BA**, or **B**. The elements cannot be repeated.

$$\left\{ \begin{array}{l} \mathbf{A} \\ \mathbf{B} \end{array} \right\} | \dots |$$

...




In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.


Δ



In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

(parameter)Δ(parameter)



The symbol  indicates a key on the keyboard. For example,  represents the carriage return key or  represents the shift key.

 *character*

 *character* indicates a control character. For example,  **Y** means that you press the control key and the **Y** key simultaneously.

base prefixes

The prefixes %, #, and \$ specify the numerical base of the value that follows:

%num specifies an octal number.

#num specifies a decimal number.

\$num specifies a hexadecimal number.

If no base is specified, decimal is assumed.

Contents

1. Data Storage	
Variable Type	1-2
Addressing Mode	1-3
The EQUIVALENCE Statement	1-4
Equivalence of Array Elements	1-4
Equivalence and Multi-Dimensioned Arrays	1-6
Equivalence Between Arrays of Different Dimensions	1-7
Equivalence of Character Variables	1-8
Equivalence in Common Blocks	1-9
Equivalence and Data Alignment	1-10
HP FORTRAN 77/iX Storage Assignment	1-11
Memory Areas	1-11
Code Area	1-11
Stack Area	1-11
Data Area	1-11
Memory Area Assignment	1-12
Local Variables	1-12
Saved Variables	1-12
Common Blocks	1-12
Summary of the Memory Area Assignment	1-13
2. Formatted Input/Output	
List-Directed Statements	2-1
List-Directed Input	2-1
List-Directed Output	2-3
Formatted Statements	2-5
Formatted Input	2-6
Formatted Output	2-7
Variable Format Descriptors	2-8
Summary of the Descriptors	2-10
Format Specifications	2-12
Integer Format Descriptors: I, O, K, @, Z	2-12
The Input Field	2-13
The Output Field	2-15
Real Format Descriptors: F, D, E, G	2-17
The Input Field	2-17
The Output Field	2-19
Character Format Descriptors: A, R	2-21
The Input Field	2-25
The Output Field	2-26
Logical Format Descriptor: L	2-30
The Input Field	2-30

The Output Field	2-30
Repeating Specifications	2-31
Correspondence Between the I/O List and Format Descriptors	2-32
Monetary Format Descriptor: M	2-34
The Input Field	2-34
The Output Field	2-35
Numeration Format Descriptor: N	2-36
The Input Field	2-36
The Output Field	2-37
Processing New Lines	2-38
The / Descriptor	2-38
The NL, NN, or \$ Descriptor	2-38
Handling Character Positions	2-39
The X Descriptor	2-39
The T Descriptor	2-40
The TL Descriptor	2-41
The TR Descriptor	2-42
Handling Literal Data	2-43
The ' and " Descriptors	2-43
The H Descriptor	2-44
Using Scale Factors: The P Descriptor	2-44
Printing Plus Signs: The S, SP, and SS Descriptors	2-49
Returning the Number of Bytes: The Q Descriptor	2-49
Terminating Format Control: The Colon Descriptor	2-50
Handling Blanks in the Input Field	2-52
The BN Descriptor	2-52
The BZ Descriptor	2-54
Alternative Methods of Specifying Input/Output	2-55
Using the Implied DO Loop	2-56

3. File Handling

Disk Files	3-1
Default File Properties	3-2
ACCESS='SEQUENTIAL'	3-2
FORM='FORMATTED'	3-2
STATUS='UNKNOWN'	3-2
Reporting File Handling Errors	3-2
The STATUS Specifier	3-3
The ERR Specifier	3-3
The IOSTAT Specifier	3-5
Creating a New File Using STATUS='NEW'	3-6
Reading From an Existing File Using STATUS='OLD'	3-9
Appending to a File	3-11
File Access	3-12
Sequential Access Files	3-12
Direct Access Files	3-14
Indexed Sequential Access Files	3-18
Unformatted I/O	3-23
Unformatted Input	3-23

Unformatted Output	3-23
Using Formatted and Unformatted Files	3-24
Using the INQUIRE Statement	3-26
Positioning the File Pointer	3-29
The BACKSPACE Statement	3-29
The REWIND Statement	3-29
The ENDFILE Statement	3-29
Example of Using the File Positioning Statements	3-30
File Handling Examples	3-31
Computing the Mean of Data in a Sequential File	3-31
Inserting Data Into a Sorted Sequential File . .	3-32
Internal Files	3-34
Reading From an Internal File	3-34
Writing to an Internal File	3-36
4. HP FORTRAN 77/iX File Operations	
The OPEN Statement Processor	4-1
Predefined Units and Files	4-2
FTN05	4-2
FTN06	4-2
FTN01 Through FTN99 (Excluding FTN05 and	
FTN06)	4-3
Creating Files with the OPEN Statement	4-4
STATUS='NEW'	4-4
STATUS='OLD'	4-4
STATUS='SCRATCH'	4-4
STATUS='UNKNOWN'	4-4
FORM='UNFORMATTED' and	
FORM='FORMATTED'	4-5
ACCESS='SEQUENTIAL'	4-5
ACCESS='DIRECT'	4-6
Closing Files	4-7
Executing the CLOSE Statement	4-7
Terminating a Program	4-7
Carriage Control Files	4-8
Terminals and Line Printers	4-8
Disk Files	4-8
FILE Equation	4-8
Using Magnetic Tapes	4-9
Using the File Handling Procedures	4-10
FSET Procedure	4-10
FNUM Procedure	4-12
UNITCONTROL Procedure	4-13

5. Subprograms	
Subroutines	5-2
Structure of a Subroutine	5-2
Invoking Subroutines	5-4
Alternate Returns From Subroutines	5-5
Functions	5-9
Function Subprograms	5-9
Statement Functions	5-15
Arguments to Subprograms	5-18
Passing Constants	5-21
Passing Expressions	5-22
Passing Character Data	5-25
Passing Arrays	5-27
Adjustable Dimensions	5-28
Assumed-Size Arrays	5-30
Multiple Entries into Subprograms	5-32
Common Blocks	5-36
Blank Common Blocks	5-36
Labeled Common Blocks	5-39
Block Data Subprograms	5-41
Using the SAVE Statement	5-43
6. Writing Efficient Programs	
Compile-Time Efficiency	6-2
Run-Time Efficiency	6-3
Declare Integer and Logical Variables Efficiently	6-3
Avoid Using Arrays	6-3
Use Efficient Data Types	6-3
Avoid Mixed-Mode Expressions	6-3
Eliminate Slow Arithmetic Operators	6-4
Use Statement Functions	6-4
Reduce External References	6-4
Combine DO Loops	6-5
Eliminate Short DO Loops	6-5
Eliminate Common Operations in Loops	6-6
Use Efficient IF Statements	6-6
Avoid Formatted I/O	6-7
Specify the Array Name for I/O	6-7
Avoid Using Range Checking	6-7
Use Your System Language	6-7
Minimize Segment Faults	6-7
MPE/iX Run-Time Efficiency Topics	6-8
Code Space Efficiency	6-9
Use Function Subroutines	6-9
Avoid Formatted I/O	6-9
Use Character Substrings	6-9
Data Space Efficiency	6-10
Eliminate Redundant or Unused Variables	6-10
Avoid Common Variables	6-10
Use INTEGER*2 and LOGICAL*2 Data	6-10
Performance Tuning	6-11

Grouping Related Routines	6-12
Shifting Data from the Data Area to the Stack Area	6-14
Integer Overflow Checking	6-14
Using the Optimizer	6-15
Introduction to the Optimizer	6-15
When to Use the Optimizer	6-15
Invoking the Optimizer on MPE/iX	6-15
Level One Optimization Modules	6-16
Branch Optimization Module	6-17
Dead Code Elimination Module	6-18
Faster Register Allocation Module	6-18
Instruction Scheduler Module	6-18
Peephole Optimization Module	6-19
Level Two Optimization Modules	6-19
Advanced Register Allocation Module	6-20
Strength Reduction	6-20
Common Subexpression Elimination	6-21
Constant Folding Module	6-21
Loop Invariant Code Motion Module	6-21
Store/Copy Optimization Module	6-22
Unused Definition Elimination Module	6-22
Optimizer Guidelines	6-23
HP FORTRAN 77 Optimizer Assumptions	6-23
OPTIMIZE Compiler Directive	6-25
Flagging Uninitialized Variables	6-30
Example	6-32
Loop Unrolling	6-35
Limits on Use	6-35
Example	6-37
What to Do If the Optimized Program Fails	6-38

7. Programming for Portability

Restricting Programs to the HP FORTRAN 77	
Standard	7-2
Using Consistent Data Storage	7-3
Use the LONG and SHORT Compiler Directives	7-3
Use Length Specifications in All Type Statements	7-4
Declare All Variables	7-5
Avoid Using the EQUIVALENCE Statement	7-5
Declare Common Blocks the Same in Every Program	
Unit	7-5
Initialize Data Before the Algorithm Begins	7-7
Avoid Accessing the Representation of Logical Values	7-7
Maintain Parameter Type and Length Consistency	7-7
Writing Programs That Can Be Easily Modified	7-9
Avoiding Unstructured FORTRAN 77 Features	7-15
Identifying Nonstandard Features	7-15
Avoiding Data Storage Inconsistencies	7-16
Using Comments	7-16
Using Conditional Compilation Directives	7-17

Resolving Incompatibilities between MPE V and	
MPE/iX: the HP3000_16	7-19
The ON Option	7-20
The OFF Option	7-20
The ALIGNMENT Option	7-21
The REALS Option	7-24
The STRING_MOVE Option	7-25
8. Interfacing with Other Languages	
HP Pascal/iX	8-2
Calling HP Pascal/iX from HP FORTRAN 77/iX	8-3
Calling HP FORTRAN 77/iX from HP Pascal/iX	8-5
HP COBOL II/iX	8-8
Calling HP COBOL II/iX from HP FORTRAN	
77/iX	8-10
Calling HP FORTRAN 77/iX from HP COBOL	
II/iX	8-11
HP C/iX	8-15
Notes on HP FORTRAN 77/iX and HP C/iX Types	8-17
Files and I/O	8-17
Parameter Passing between HP FORTRAN 77 and	
HP C	8-18
Using System Intrinsic	8-20
Defining System Intrinsic	8-20
Matching Actual and Formal Parameters	8-21
9. Debugging FORTRAN 77 Programs	
Using xdb	9-1
The Strategy	9-3
Program Requirement	9-3
Linker and Debugger Interaction	9-4
Invoking the Debugger	9-4
Exiting the Debugger	9-5
Executing Your Program	9-5
Viewing the Execution Stack	9-5
Viewing the Source File	9-6
The View Command	9-6
The Window Command	9-7
The Move Command	9-7
The Search Commands	9-7
Viewing Program Data	9-7
Listing the Variables	9-7
Finding a Variable's Value	9-9
Execution Control	9-9
Breakpoints	9-9
Setting Breakpoints	9-10
Recovering from Breakpoints	9-10
Deleting Breakpoints	9-11
Using Breakpoints for Execution Tracing	9-11
Single Step Commands	9-12
Additional Debugger Capabilities	9-12

Removing Debugging Information	9-12
Using HP Toolset/iX	9-13
Compiling Programs for HP Toolset/iX	9-13
Invoking HP Toolset/iX	9-13
Setting Up for Symbolic Debug	9-13
When to Use HP Toolset/iX	9-14
Running a Program	9-14
Setting Breakpoints	9-14
Tracing Names	9-15
Clearing Breakpoints	9-16
Displaying Variables	9-16
Modifying Variables	9-18
Redoing a Command	9-19
Restarting Your Program	9-19
Displaying Breakpoints	9-19
Using the Trace Facilities	9-20
Accessing MPE/iX Debug	9-20
Ending Execution of a Program Prematurely	9-20
Exiting HP Toolset/iX	9-20

Index

Figures

2-1. The <i>Aw</i> Format Descriptor	2-23
2-2. The <i>Rw</i> Format Descriptor	2-23
2-3. The <i>Aw</i> and <i>Rw</i> Format Descriptors	2-24
3-1. Sequential Access Files	3-13
3-2. Direct Access Files	3-14
3-3. Indexed Sequential Access Files	3-18
7-1. MPE V Structure	7-22
7-2. MPE/iX Structure	7-22
7-3. Shifted MPE/iX Structure	7-23

Tables

1-1. Data Type Keywords	1-2
1-2. Addressing Mode	1-3
1-3. Data Class and Addressing Mode	1-3
1-4. Summary of Memory Area Assignment	1-13
2-1. Summary of the Format Descriptors	2-10
2-2. Summary of the Edit Descriptors	2-11
3-1. Status Types	3-3
3-2. The STATUS Specifier	3-3
3-3. The IOSTAT Specifier	3-5
4-1. UNITCONTROL Options	4-13
5-1. Components of Program Units	5-1
5-2. Categories of FORTRAN Functions	5-9
6-1. Data Type Efficiency	6-3
6-2. Descriptions of Assembly Language Routines	6-16
7-1. Conditional Compilation Directives	7-17
7-2. HP3000_16 Directive Options	7-20
7-3. Data Alignment on MPE V and MPE/iX	7-21
8-1. HP FORTRAN 77/iX and HP Pascal/iX Types	8-2
8-2. HP COBOL II/iX Numeric Types and Formats	8-8
8-3. HP COBOL II/iX and HP FORTRAN 77/iX Data Types	8-9
8-4. HP FORTRAN 77/iX and HP C/iX Types	8-15
8-5. HP FORTRAN 77/iX and HP Pascal/iX Data Types	8-21
9-1. Sample xdb Commands	9-2

Data Storage

The data allocation for a FORTRAN variable is influenced by three factors:

- Variable type
- Addressing mode
- The EQUIVALENCE statement

This chapter describes these factors in detail.

Variable Type

The variable data types of FORTRAN 77 and their corresponding assignment statements are shown in Table 1-1.

Table 1-1. Data Type Keywords

General Name	Data Type Keyword	Equivalent Keyword
Integer	INTEGER*2 ² INTEGER*4 ²	INTEGER ^{1,3} (option) INTEGER ^{1,3} (default)
Real	REAL*4 ² REAL*8 ² REAL*16 ²	REAL ¹ DOUBLE PRECISION ¹ (none)
Complex	COMPLEX*8 ² COMPLEX*16 ²	COMPLEX ¹ DOUBLE COMPLEX ¹
Logical	LOGICAL*1 ² LOGICAL*2 ² LOGICAL*4 ²	BYTE ² LOGICAL ^{1,3} (option) LOGICAL ^{1,3} (default)
Character	CHARACTER ¹	(none)

Notes

1. ANSI 77 standard.
2. Extension to the ANSI standard.
3. The equivalence depends on the setting of the compiler directives LONG and SHORT.

The size of each data type determines how much storage is allocated to the variable. In addition to the size, the alignment requirement of each type determines where the variable begins in storage. For example, a variable of CHARACTER*7 is allocated seven bytes of storage, starting on a byte boundary. Similarly, INTEGER*2 allocates two bytes of storage, starting on a four byte boundary. The size of each data type is the same on different systems, but the alignment requirement is usually system dependent. Refer to the *HP FORTRAN 77/iX Reference* for more details on the format and alignment for each data type.

Addressing Mode

The FORTRAN compiler generates machine instructions that access program data. Depending on the type of the data, the instructions use any of the modes of addressing shown in Table 1-2.

Table 1-2. Addressing Mode

Addressing Mode	Description
Direct	Accesses data directly by using the address given to the variable.
Indirect	Accesses data by using a pointer to the address of the data.
Descriptor	Accesses data by using a record containing the address of the data and the maximum length.

Direct addressing saves both storage and time. Indirect addressing requires an extra pointer in addition to the regular variable storage. Descriptor addressing requires multiple words, including a pointer and the maximum length of the data item; the number of words is machine dependent. The access mode is determined by the data class, as summarized in Table 1-3.

Table 1-3. Data Class and Addressing Mode

Data Class	Addressing Mode
Common variables.	Indirect
Static variables (variables in a SAVE or DATA statement).	Indirect
Variables in an EQUIVALENCE statement.	Direct and indirect
Local variables not in a SAVE or DATA statement.	Direct
Parameters: Character string parameters Other parameters	By descriptor Indirect
Other variables	Direct

Note



Make sure all variables are properly initialized. HP Link Editor/iX does not initialize all the stack space as the Segmenter on MPE V does. Uninitialized variables that do not cause problems on MPE V/E-based systems might cause programs to abort on MPE/iX-based systems.

HP FORTRAN 77/V stores variables larger than eight bytes indirectly; HP FORTRAN 77/iX stores the variables directly.

The EQUIVALENCE Statement

Storage space is allocated in memory consecutively; every variable is independent. Except for common variables, the declaration order of variables in the source code does not determine the order in which the variables are allocated in memory. However, this pattern of allocation can be changed with the EQUIVALENCE statement, which allows overlapping of the same storage space with more than one variable. Care must be taken when data types of different sizes share the same storage space. This section describes how to use the EQUIVALENCE statement in situations that require special attention.

Equivalence of Array Elements

Array elements can share the same storage space with elements of a different array or with simple variables. For example, the statements:

```
REAL*4 a(3), c(5)
EQUIVALENCE (a(2), c(4))
```

specify that array element a(2) shares the same storage space as array element c(4). This implies that:

- a(1) shares storage space with c(3)
- a(3) shares storage space with c(5)
- No equivalence occurs outside the bounds of the arrays

The storage space for the two arrays is shown in the following table:

Array a	Storage Space Byte Number	Array c
	1-4	c(1)
	5-8	c(2)
a(1)	9-12	c(3)
a(2)	← 13-16 →	c(4)
a(3)	17-20	c(5)

By using the EQUIVALENCE statement, array elements can share the same storage space. If the arrays are not of the same type, they might not line up element-by-element. For example, the statements:

```
REAL*4 a(2)
INTEGER*2 ibar(4)
EQUIVALENCE (a(1), ibar(1))
```

produce the following storage space allocation:

Array a	Storage Space Word Number	Array ibar
a(1)	1 - 4	ibar(1) ibar(2)
a(2)	5 - 8	ibar(3) ibar(4)

Placing an array name only in an EQUIVALENCE statement has the same effect as using an array element name that specifies the first element of the array. That is, the statement:

```
EQUIVALENCE (a,ibar)
```

produces the same results as:

```
EQUIVALENCE(a(1), ibar(1))
```

When array elements share the same storage space with other array elements or variables, the same storage space cannot be occupied by more than one element of the same array. For example, the statements:

```
DIMENSION a(2)
EQUIVALENCE (a(1),b), (a(2), b)
```

are illegal because they specify the same storage space for a(1) and a(2).

An EQUIVALENCE statement must not specify that consecutive array elements are noncontiguous. For example, the statements:

```
REAL a(2), r(3)
EQUIVALENCE (a(1), r(1)), (a(2), r(3))
```

are illegal because the EQUIVALENCE statement specifies that a(1) and a(2) are noncontiguous.

Equivalence and Multi-Dimensioned Arrays

As an extension to the ANSI standard, you can indicate on an EQUIVALENCE statement the element of a multi-dimensioned array by specifying its position in the array. For example, the statements:

```
INTEGER*4 total (3,2)
INTEGER*4 sum (6)
EQUIVALENCE (sum, total(1))
```

produces the following storage space allocation:

Array Element	Equivalent Array Element
total(1,1)	sum(1)
total(2,1)	sum(2)
total(3,1)	sum(3)
total(1,2)	sum(4)
total(2,2)	sum(5)
total(3,2)	sum(6)

According to the ANSI standard, an element of a multi-dimensioned array must be referenced by one subscript for each dimension. However, using the equivalence and multi-dimensioned array feature, you can specify one subscript (in EQUIVALENCE statements only) to indicate a position in memory independent of an array's declared number of dimensions.

Equivalence Between Arrays of Different Dimensions

To determine equivalence between arrays with different dimensions, FORTRAN contains an internal array successor function that views all elements of an array in linear sequence. Each array is stored as if it were a one-dimensional array. Array elements are stored in ascending sequential column-major order. The first index varies the fastest, then the second, and so on. For example, the array:

```
i(-2:4)
```

stores the elements of array *i* in this order:

```
i(-2) i(-1) i(0) i(1) i(2) i(3) i(4)
```

The array:

```
t(2,3)
```

stores the elements in the following order:

```
t(1,1) t(2,1) t(1,2) t(2,2) t(1,3) t(2,3)
```

Similarly, the array:

```
k(2,2,3)
```

stores the elements in the following order (reading left to right and top to bottom by row):

```
k(1,1,1) k(2,1,1) k(1,2,1) k(2,2,1)
k(1,1,2) k(2,1,2) k(1,2,2) k(2,2,2)
k(1,1,3) k(2,1,3) k(1,2,3) k(2,2,3)
```

The number of bytes each element occupies depends on the type of the array. For example, the statements:

```
REAL*4      a
INTEGER*2   i
DIMENSION  a(2,2), i(4)
EQUIVALENCE (a(2,1), i(2))
```

produce the following storage space allocation:

Array a	Storage Space Byte Number	Array i
a(1,1)	1 - 2 3 - 4	i(1)
a(2,1)	5 - 6 7 - 8	i(2) i(3)
a(1,2)	9 - 10 11 - 12	i(4)
a(2,2)	13 - 16	

Equivalence of Character Variables

As an extension to the ANSI 77 Standard, character and noncharacter data items can share the same storage space. For example, the statements:

```
INTEGER*4 i(5)
CHARACTER*16 c
EQUIVALENCE(i,c)
```

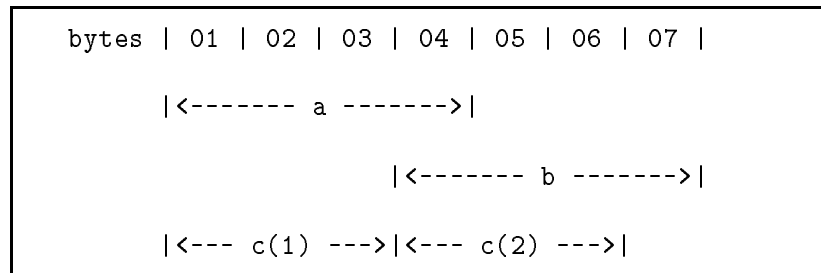
produce the following storage space allocation:

Array i	Storage Space Byte Number	Variable c
i(1)	1 - 4	c(1:4)
i(2)	5 - 8	c(5:8)
i(3)	9 - 12	c(9:12)
i(4)	13 - 16	c(13:16)
i(5)	17 - 20	

The lengths of the data items that share the same storage space do not have to match. An EQUIVALENCE statement specifies that the storage sequence of the character data items whose names are specified in the list have the same first character storage unit. This causes the association of the data items in the list and can cause association of other data items. Any adjacent characters in the associated data items can also have the same character storage unit and therefore can also be associated. For example, the statements:

```
CHARACTER*4 a, b
CHARACTER*3 c(2)
EQUIVALENCE (a,c(1)), (b,c(2))
```

cause association between a, b, and c in this way:



Equivalence in Common Blocks

Data elements can be put into a common block by specifying the elements as equivalent to data elements mentioned in a COMMON statement. If one element of an array shares the same storage space with a data element in a common block by using the EQUIVALENCE statement, the whole array is placed in the common block. Equivalence is maintained for the storage unit preceding and following the data element in common.

When necessary, the common block is extended to fit an equivalenced array into the common block. However, no array can be put into a common block with the EQUIVALENCE statement if storage elements have to be prefixed to the common block to contain the entire array.

Equivalences cannot insert storage into the middle of the common block or rearrange storage within the block. Because the elements in a common block are stored contiguously in the order they are listed in the COMMON statement, two elements in common cannot be made to share the same storage space with the EQUIVALENCE statement. For example, in the statements:

```
INTEGER*4 i(6), j(6)
COMMON i
EQUIVALENCE (i(3), j(2))
```

the array *i* is in a common block and array element *j* is equivalent to *i*(3).

The common block is extended to accommodate array *j* as follows:

Array <i>i</i>	Storage Space Byte Number	Variable <i>j</i>
<i>i</i> (1)	1 - 4	
<i>i</i> (2)	5 - 8	<i>j</i> (1)
<i>i</i> (3)	9 - 12	<i>j</i> (2)
<i>i</i> (4)	13 - 16	<i>j</i> (3)
<i>i</i> (5)	17 - 20	<i>j</i> (4)
<i>i</i> (6)	21 - 24	<i>j</i> (5)
	25 - 28	<i>j</i> (6)

The equivalence set up by the statements:

```
INTEGER*4 i(6), j(6)
COMMON i
EQUIVALENCE (i(1), j(2))
```

is not allowed. To set array *j* into the common block, four extra bytes must be inserted in front of the common block and element *j*(1) would be stored in front of the common block.

Equivalence and Data Alignment

Each data type has its own data alignment requirement that is system dependent; see the *HP FORTRAN 77/iX Reference* for details. If you force any variable to start on a boundary other than the alignment required, a compilation error occurs. For example, if the data alignment of character variables is on any byte boundary and the data alignment of INTEGER*2 variables is on even byte (16-bit) boundaries, the following is illegal:

```
INTEGER*2 i, j(2)
CHARACTER*6 c
EQUIVALENCE (c, i)
EQUIVALENCE (c(2:2), j)
```

In these statements, either *i* or *j* would have to start on an odd byte boundary, which violates the alignment requirement. However, if the equivalence of *c* and *i* is removed, the statements:

```
INTEGER*2 i, j(2)
CHARACTER*6 c
EQUIVALENCE (c(2:2), j)
```

produce the following storage space allocation:

Array j	Storage Space Byte Number	Variable c
	0 - 1	1 unused byte, c(1:1)
j(1)	2 - 3	c(2:3)
j(2)	4 - 5	c(4:5)
	6 - 7	c(6:6), 1 unused byte

Note that *c* starts on an odd byte boundary and *j* starts on an even byte (16-bit word) boundary.

HP FORTRAN 77/iX Storage Assignment

This section describes the 900 Series HP 3000 computer hardware architecture as it relates to the placement of FORTRAN 77 data objects. This section also describes the actions of the compiler to assign data areas to FORTRAN 77 variables and common blocks. Refer to the *HP FORTRAN 77/iX Reference* for information on data formats of the FORTRAN 77 data types.

Memory Areas

A compiled program uses the following memory areas:

- Code
- Stack
- Data

Each area has different characteristics, as described below.

Code Area

The code area is used for the machine instructions and constants generated by the compiler. One code area is generated for each program. The maximum size of a code area is 2^{30} bytes.

Stack Area

The stack area is used for any variables local to a particular routine and for any variables that are passed from one routine to another. Note the following about stack areas:

- The size of the stack area is system dependent. For MPE/iX, the maximum size of the stack and data area combined is 1 073 741 824 bytes.
- The stack area must be in memory for the program to execute.
- Access to the local variables in the stack area is faster than the access to passed parameters.

Data Area

The data area is normally used for variables that are in common, saved variables, and initialized variables. In general, accessing data from this area requires two instructions, as opposed to one instruction for accessing data from the stack area.

The stack area is dynamically allocated; thus variables that must be accessed from multiple subprograms (such as those in common) and variables that must retain their values across multiple invocations of a subprogram (such as those that are saved) must be placed in the data area.

The size of the data area is system dependent. For MPE/iX, the maximum size of the stack and data area combined is 1 073 741 824 bytes.

Memory Area Assignment

This section describes how variables and common blocks are assigned to the memory areas.

Note



The location field from the TABLES ON compiler directive lists each data object's assigned memory area. Use this directive if there is any doubt about the location of a data object.

Local Variables

Uninitialized local variables are normally assigned to the stack area.

Variables that are initialized with a DATA statement, such as

```
DATA var /3.2/
```

are assigned to the data area. This allows the compiler to preassign values to these variables statically at link time. If the variables were placed in the stack area, they would have to be reinitialized each time the subprogram in which they were declared was entered.

Saved Variables

When a local variable is specified in a SAVE statement, such as

```
SAVE var
```

the variable is assigned to the data area, not to the stack area. This allows the variable to retain its value over multiple invocations of the subprogram in which it is declared. When you specify

```
$SAVE_LOCALS [ON]
```

all local variables are saved and thus are moved from the stack area to the data area.

Common Blocks

The compiler places both named and unnamed common blocks in the data area. The Link Editor matches the names of the common blocks declared in different subroutines or functions and determines where to place the blocks in the data area.

Summary of the Memory Area Assignment

Table 1-4 summarizes the assignment rules of the memory areas.

Table 1-4. Summary of Memory Area Assignment

Description	Code Area	Stack Area	Data Area
Local variables		X	
Variables with \$SAVE_LOCALS specified			X
Variables specified with a SAVE statement			X
Variables specified with a DATA statement			X
Common blocks			X
Formats			X
Constants	X		
Machine instructions	X		
Compiler temporaries		X	
Passed addresses		X	

Formatted Input/Output

HP FORTRAN 77 provides you with control over I/O operations through the use of format specifications. Formatted I/O can be used for I/O access to devices (such as terminals), line printers, or disk files. This chapter describes how to use formatted I/O and list-directed (free-format) I/O. Chapter 3 describes how to use unformatted I/O and nonstandard files.

An I/O statement is connected to its source or destination through a unit number specified as a parameter in the READ or WRITE statement. All HP FORTRAN 77 programs have at least two “preconnected” unit numbers: 5 and 6. These unit numbers are automatically connected to the standard input and output files, respectively. Typically, your terminal is the default standard input and output file.

List-Directed Statements

List-directed I/O, sometimes called “free format” I/O, is the simplest kind of formatted I/O. List-directed I/O lets the compiler select a format for the I/O data, depending upon the type and magnitude of the data in the variable list on the I/O statement (therefore the name list-directed). List-directed I/O is selected by an asterisk where the format specification normally appears.

List-Directed Input

A list-directed READ statement can be written in one of two ways. First, the READ statement can specify the input unit number for the source of the input data. For example,

```
READ(5,*) i, j, k
```

Second, the READ statement does not have to specify the unit number; instead, the input data is taken from the standard input device. For example,

```
READ *, i, j, k
```

Data items read by a list-directed READ statement can be separated by a comma, blanks, or can be on separate lines. This flexibility makes list-directed input convenient for you to enter data.

The input field can be terminated abruptly by entering a slash (/) in the input field, causing any remaining items in the I/O list to be skipped. For example, if the input line:

```
5 /
```

is read by one of these statements:

```
READ(5,*) i, j, k
```

```
READ *, i, j, k
```

the variable `i` is assigned the value 5, and the read terminates. The variables `j` and `k` are unchanged. Entering a slash is useful when you want to enter only the first few values of a long input list.

List-directed input data can contain a multiplier to enter many copies of an input value. For example, the input line:

```
3*1024
```

assigns the value of 1024 to three input variables.

Character data read with a list-directed `READ` statement must be enclosed in quotation marks if the data contains any of the following separators: blank (), comma (,) or slash (/). This is because list-directed input uses blanks, commas, and slashes as data separators. (If the input field is specified by format descriptors, apostrophes are not required.) For example, the statements:

```
INTEGER*4 id, section  
CHARACTER*10 name  
READ *, id, name, section
```

accept the following input string:

```
2612 'J. Smith' 7
```

As an HP extension to the FORTRAN 77 standard, data can be read from an internal file with a list-directed `READ`.

List-Directed Output

A list-directed output statement can be written in one of two ways. First, the WRITE statement must have a unit number for the destination of the output data. For example,

```
WRITE(6,*) 'Output values=', i, j, k
```

Second, the PRINT statement always writes to the standard output device. For example,

```
PRINT *, 'Output values=', i, j, k
```

There is no WRITE statement equivalent to the PRINT statement form. That is, this statement is illegal:

```
WRITE *, i, j, k
```

You can include a comma before the list of data items in the WRITE statement, as shown below:

```
WRITE(6,*), a, b, c
```

List-directed output prints numeric data with a leading blank. Character data is printed without any leading blanks.

Numeric data can be printed in scientific notation, depending upon the magnitude. However, you should not use list-directed output for applications where the exact format of the output data is critical.

The program below illustrates the two ways of writing list-directed input and output statements.

```
PROGRAM list_directed_io
```

```
C List-directed input statements:
```

```
READ(5,*) i1          ! Input from the preconnected input unit.  
READ *, i2           ! Input from the standard input device.
```

```
C List-directed output statements:
```

```
WRITE(6,*) 'i1=', i1  ! Output to the preconnected output unit.  
PRINT *, 'i2=', i2   ! Output to the standard output device.
```

```
END
```

Following is another program with list-directed output statements:

```
PROGRAM output_ex

COMPLEX vector
CHARACTER string*8

vector = (1.0, 1.0)
string = 'alphabet'
int = 123
var = 123.456E29

PRINT *, vector, string, int, var

END
```

The output of this program is the following line:

```
(1.0,1.0)alphabet 123 1.23455E+31
```

Note that blanks are inserted before numeric values and not before character strings. List-directed I/O can also be done on nonstandard files, as described in Chapter 3. In particular, list-directed WRITE statements can use internal files.

Formatted Statements

Formatted I/O statements use format descriptors that supplement the READ, WRITE, or PRINT statements to exactly define the format of the data. The descriptors can be specified on the READ, WRITE, or PRINT statements, or can appear in a FORMAT statement.

The FORMAT statement is a nonexecutable statement that describes how the data listed in a READ, WRITE, or PRINT statement is to be arranged. The FORMAT statement can appear anywhere in a program after a PROGRAM, FUNCTION, or SUBROUTINE statement.

The type of conversion indicated by the format descriptors should correspond to the data type of the variable in the I/O list. The format descriptors are described later in this chapter.

Formatted Input

The formatted READ statement transfers data from an external device to internal storage, and converts the ASCII data to internal representation according to the format descriptor.

One way of specifying a formatted READ statement is to place the descriptors on the READ statement itself. For example, the statement:

```
READ (5, '(I5, F5.1)') int, value
```

reads data from unit 5 into `int` and `value` according to the format descriptors I5 and F5.1, respectively.

If there are many format descriptors or if the same descriptors are used repeatedly, place the descriptors in a FORMAT statement. To specify a FORMAT statement, use one of the following forms of a formatted READ statement. The statements:

```
READ (5,100) a, b, c
100 FORMAT (F7.1, F8.1, F9.1)
```

read data from unit 5 into the variables `a`, `b`, and `c` according to the format descriptors F7.1, F8.1, and F9.1, respectively.

The statements:

```
READ 100, a, b, c
100 FORMAT (F7.1, F8.1, F9.1)
```

get data from the standard input file according to the format descriptors in the FORMAT statement labeled 100.

The following program shows several ways of writing formatted input statements.

```
PROGRAM formatted_input
```

```
C Formatted input statements from the preconnected input unit:
```

```
READ(5, '(I9)') i

READ(5, 100) i
100 FORMAT(I9)
```

```
C Formatted input statements from the standard input device:
```

```
READ(*, '(I9)') i

READ(*, 100) i      ! Note: These statements reuse the
READ 100, i        ! FORMAT statement labeled 100 above.

READ '(I9)', i

END
```

Formatted Output

The formatted WRITE and PRINT statements transfer data from the storage location of the variables or expressions named in the output list to the file associated with the specified unit. The data is converted to a string of ASCII characters according to the format descriptors.

One way of specifying a formatted WRITE statement is to include the format descriptors on the WRITE statement itself. For example, the statement:

```
WRITE (6, '(1X, I5, F3.1)') int, value
```

writes the values of `int` and `value` to unit 6 (the preconnected output unit) according to the format descriptors I5 and F3.1, respectively. The statement:

```
PRINT '(1X, I5, F3.1)', int, value
```

also writes data from `int` and `value` to the standard output device according to the format descriptors I5 and F3.1.

To use a FORMAT statement, specify its label in the corresponding WRITE or PRINT statements. For example, the statements:

```
WRITE (6,100) int, value
100 FORMAT (1X, I5, F3.1)
```

write data from variables `int` and `value` to the preconnected output unit according to the descriptors in the FORMAT statement labeled 100.

The following program shows various ways of writing formatted output statements.

```
PROGRAM formatted_output
i = 15
```

C Formatted output statements to preconnected output unit:

```
WRITE(6, '(1X, "i=", I9)') i      ! The output of these
WRITE(6,200) i                  ! statements looks the
200 FORMAT (1X, 'i=', I9)       ! same.
```

C Formatted output statements to standard output device:

```
WRITE(*, '(1X, "i=", I9)') i
WRITE(*, 200) i
```

C Note: This statement reuses the FORMAT statement labeled 200 above

```
PRINT '(1X, "i=", I9)', i
PRINT 200, i
```

C Note: This statement reuses the FORMAT statement labeled 200 above
END

Variable Format Descriptors

Variable format descriptors allow the values of integer variables, integer constants, and character constants to be imported into format strings. Integer variable format descriptors may be used wherever an integer may appear, except to specify the number of characters in a Hollerith field. To use a variable format descriptor, enclose the variable or constant in angle brackets. The following is an example of a variable format descriptor:

```
FORMAT (I<isize>)
```

In this example, the FORMAT statement performs an I (integer) data transfer with a field width equal to the value of `isize` when the format is scanned.

Variables may be `INTEGER*2` or `INTEGER*4`. The value of a variable format descriptor must be of a valid magnitude for its use in the format; otherwise an error occurs.

Variable format descriptors are not allowed in run-time formats (that is, those that are assembled in arrays or character expressions at run-time).

If a variable is used, its value is reevaluated each time it is encountered in the normal format scan. If the value of a variable used in a descriptor changes during execution of the I/O statement, the new value is used the next time the format item containing the descriptor is processed.

The following example program illustrates the use of variable format descriptors.

C Program illustrating variable format descriptors.

```
PROGRAM varfmt1
INTEGER n
PARAMETER (n = 3)
REAL x(n,n)
DATA x / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /
```

C Print out the constants 1 through 3 in variable width fields.

```
DO 10 j = 1,3
PRINT 100,j
100 FORMAT (1x, I<j>)
10 CONTINUE
```

C Print out the lower diagonal elements of matrix x.

```
DO 20 i = 1,n
PRINT 101, (x(i,k), k = 1,I)
101 FORMAT (1x, <I>F5.1)
20 CONTINUE
END
```

The output of of the program is as follows:

```
1
2
3
1.0
2.0 5.0
3.0 6.0 9.0
```

Summary of the Descriptors

The format descriptors, which describe the data, are summarized in Table 2-1.

Table 2-1. Summary of the Format Descriptors

Data Conversion Type	Format Descriptor	Forms	Data Declarations Allowed
Character	A R	A[<i>w</i>] R[<i>w</i>]	All data types All data types
Logical	L	L[<i>w</i>]	LOGICAL
Real	D E F G	D[<i>w.d</i>] E[<i>w.d</i> [E <i>e</i>]] F[<i>w.d</i>] G[<i>w.d</i> [E <i>e</i>]]	All numeric data types All numeric data types All numeric data types All numeric data types
Integer	I	I[<i>w.m</i>]	All numeric data types
Monetary	M	M[<i>w.d</i>]	All numeric data types
Numeration	N	N[<i>w.d</i>]	All numeric data types
Octal	O K @	O[<i>w.m</i>] K[<i>w.m</i>] @[<i>w.m</i>]	Input: INTEGER Output: All data types
Hexadecimal	Z	Z[<i>w.m</i>]	Input: INTEGER Output: All data types

In the table above,

- w* is the field width †
- d* is the digits to the right of the decimal point †
- m* is the minimum digits to be output (if omitted, *m* = 1)
- e* is the number of digits of the exponent (if omitted, *e* = 2)

† The default values are described in the *HP FORTRAN 77/iX Reference*.

Note



In Table 2-1:

“All numeric data types” implies LOGICAL*1, LOGICAL*2, LOGICAL*4, INTEGER*2, INTEGER*4, REAL*4, REAL*8, REAL*16, COMPLEX*8, and COMPLEX*16.

“INTEGER” implies INTEGER*2 and INTEGER*4.

“LOGICAL” implies LOGICAL*1, LOGICAL*2, and LOGICAL*4.

The edit descriptors control the positioning and formatting of numeric, Hollerith, and logical fields on input and output lines. The edit descriptors do not cause data conversions and, with the exception of the Q descriptor, are not associated with the variables on the READ, WRITE, or PRINT statements. The edit descriptors are summarized in Table 2-2, where n represents a positive, nonzero number.

Table 2-2. Summary of the Edit Descriptors

When Used	Edit Descriptor	Descriptor Type	Description
Input	BN	Numeric	Ignore blanks in input field
	BZ	Numeric	Treat blanks as zeros
	Q	Integer	Returns the number of remaining bytes on the input record
Output	NL	Prompt	Cursor moves to a new line†
	NN or \$	Prompt	Cursor remains on the same line†
	S	Numeric	Plus sign (+) suppressed
	SP	Numeric	Plus sign (+) printed
	SS	Numeric	Plus sign (+) suppressed
	"	Character	Writes character constant
	'	Character	Writes character constant
Input/Output	nP	Scale factor	Modifies input/output of the $Ew.d$, $Dw.d$, and $Gw.d$ descriptors and output of the $Fw.d$ descriptor
	nX	Position edit	Skips n positions
	Tn	Tab edit	Positions to column n
	TLn	Tab edit	Positions backward n columns
	TRn	Tab edit	Positions forward n columns
	:	Format control	Terminates format if no more items are in the I/O list
	/	Line terminator	Begins processing a new line

† See the *HP FORTRAN 77/iX Reference* for details.

The input descriptors are ignored on output. The output descriptors are ignored on input, except that nH , $"$, and $'$ are treated as nX , where n is the length of the string.

Format Specifications

This section describes the format and edit descriptors in detail.

Note



In the examples for this chapter, Δ represents a blank space and \sqcup represents eight binary zeros.

Integer Format Descriptors: I, O, K, @, Z

The I, O, K, @, and Z format descriptors provide formatting for integer data types. The general specifications are:

I[*w*.[*m*]]

O[*w*.[*m*]]

K[*w*.[*m*]]

@[*w*.[*m*]]

Z[*w*.[*m*]]

where *w* is the width of the field, and *m* is the minimum number of digits to be output; if *m* is not used, a value of 1 is assumed. The I*w.m* form is only used for output; on input, the *m* is ignored.

The Input Field

The *Iw* format descriptor interprets the next *w* positions of the input record. You can omit the plus sign for positive integers; you must not have a decimal point in the input record.

The input statement:

```
READ(5, '(I3)') int
```

can read any of the following input values:

Input Field	Equivalent Assignment
12	int = 12
+12	int = 12
-12	int = -12
Δ123	int = 12
+123	int = 12
-123	int = -12
123456	int = 123
↑ First column of input field	

The O, K, and @ input field can have up to 11 octal digits; the O, K, and @ descriptors are interchangeable. The octal digits are: 0, 1, 2, 3, 4, 5, 6, and 7; plus or minus signs are not allowed. If any nondigit appears, (other than a blank), an error occurs. The variable receiving the octal value must be an INTEGER*2 or INTEGER*4 data type.

The input statement:

```
READ(5, '(O3)') ioctal
```

can read any of the following input values:

Input Field	Equivalent Assignment
123456	ioctal = 83 (decimal equivalent of 123 octal)
1234	ioctal = 83
123	ioctal = 83
12	ioctal = 10 (decimal equivalent of 12 octal)
↑ First column of input field	

The Z input field contains these hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A or a, B or b, C or c, D or d, E or e, and F or f. If the number of digits is too long for the integer variable, undefined results occur. If a nonhexadecimal digit is used, an error occurs; no leading plus or minus sign on input is allowed. The variable receiving the hexadecimal input must be an INTEGER*2 or INTEGER*4 data type.

The input statement:

```
READ(5, '(Z3)') ihex
```

can read any of the following input values:

Input Field	Equivalent Assignment
12abcd	ihex = 298 (decimal equivalent of 123 hex)
12ab	ihex = 298
12a	ihex = 298
12	ihex = 18 (decimal equivalent of 12 hex)
1	ihex = 1 (decimal equivalent of 1 hex)
↑	
First column of input field	

The Output Field

The I, O, K, @, and Z descriptors each produce a distinctive output format.

The I and Z Output Format

The I and Z format descriptors handle the output field in the same way. The I descriptor writes numbers in an integer format; the Z descriptor writes hexadecimal data. For the I or Z descriptor, if the field width w is smaller than the number of digits needed to represent the value, the field is filled with asterisks.

If the field length w is greater than the length of the integer value, the integer is output right-justified in the field, with blanks on the left.

For integer values, the output of a negative number requires a print position for the negative sign. If the $Iw.m$ form is used and the output value is less than m positions, the value is preceded by zeros. If m equals zero, a 0 value is output as all blanks.

For the Z descriptor, the optional m value specifies a minimum number of digits to be output, forcing leading zeros as necessary up to the first non-zero digit.

The O, K, and @ Output Format

The O, K, and @ format descriptors write octal data. The output field can have up to 6 octal digits for INTEGER*2 data and can have up to 11 digits for INTEGER*4 data. The octal digits are: 0, 1, 2, 3, 4, 5, 6, and 7; plus or minus signs are not displayed.

If the field width w is smaller than the length of the integer value, the field is filled with asterisks.

If the field length w is greater than the digits in the integer value, the integer is output right-justified in the field, with zeros on the left.

The optional m value specifies a minimum number of digits to be output, forcing leading zeros as necessary up to the first non-zero digit.

The following program compares the output generated by the I, O, and Z formats.

```
PROGRAM int_outputs

INTEGER*4 int
int = 12

WRITE(6, '(11X, "I9", 15X, "011", 15X, "Z9")')

DO i = 1, 9
  WRITE(6,100) int, int, int
  IF (i .NE. 9) int = int * 10
END DO

100 FORMAT (6X, "{", I9, "}", 6X, "{", O11, "}", 6X, "{", Z9, "}")
STOP
END
```

The output from the program is as follows:

I9	O11	Z9
{ 12}	{ 14}	{ C}
{ 120}	{ 170}	{ 78}
{ 1200}	{ 2260}	{ 4B0}
{ 12000}	{ 27340}	{ 2EE0}
{ 120000}	{ 352300}	{ 1D4C0}
{ 1200000}	{ 4447600}	{ 124F80}
{ 12000000}	{ 55615400}	{ B71B00}
{120000000}	{ 711607000}	{ 7270E00}
{*****}	{10741506000}	{ 47868C00}

Real Format Descriptors: F, D, E, G

The F, D, E, and G format descriptors provide formatting for REAL*4, REAL*8, REAL*16, COMPLEX*8 and COMPLEX*16 data types. Complex values are treated as pairs of real values. The general specifications are:

F[*w.d*]

E[*w.d*[E*e*]]

D[*w.d*]

G[*w.d*[E*e*]]

where *w* is the total field width, *d* is the number of digits after the decimal point, and *e* is the number of digits for the exponent.

The Input Field

The F, D, E, and G format descriptors handle the input field in the same way. The input field can consist of a REAL*4, REAL*8, or REAL*16 number in either floating-point or exponential form. If a decimal point is not supplied, it will be inserted *d* digits from the rightmost digit.

The F descriptor is used most often, although any one of the format descriptors can be used for input of real data types.

The input statement

```
READ(5, '(F6.2)') a
```

can read any of the following input values:

Input Field	Equivalent Assignment
Δ123	a = 1.23
123456	a = 1234.56
12345678	a = 1234.56
+12345678	a = 123.45
12.345	a = 12.345
-123.45	a = -123.4
1234E3*	a = 12340.0
123E3	a = 1230.0
123E-3	a = 0.00123
12.E-3	a = 0.120
1234D3	a = 0.1234D+05
123.D3	a = 0.123D+06
blanks	a = 0.0
↑ First column of input field	

* 1234E3 → 1234000 → 12340.00

If **Ee** falls within **w** field (**Fw.d**) then the number is expanded and the decimal is placed before the second digit from the right.

- 1) 1234E24 --> 1234E2 --> 123400 --> 1234.00
- 2) 1234E8 --> 1234E8 --> 1234 x 10⁸ --> 1234 x 10⁶ --> 1234E+9
- 3) 12345E8 --> 12345E --> 12345 --> 123.45

The Output Field

The F, E, D, and G descriptors each produce a distinctive output format.

The F Output Format

The F format descriptor writes numbers in a fixed-point format. That is, the decimal point can be fixed d places from the right of the number. The field width w should always be at least two greater than the total number of digits you want printed; this leaves room for a sign and a decimal point.

The E Output Format

The E format descriptor writes any floating-point type numbers in exponential format. The number is printed in normalized floating-point format; that is, the decimal point is moved to the left of the number. The letter E is printed before the exponent, which consists of a sign and two digits. The optional e specification in the $Ew.dEe$ format changes the field width allocated for the exponent. The field width is two places if Ee is omitted.

The D Output Format

The D format descriptor writes any floating-point type numbers in exponential format. The D descriptor is the same as the E format descriptor. On D output format, the exponent character will always be the letter E.

The G Output Format

The G format descriptor writes numbers in either floating-point or exponential format, depending on the size of the number. The $Gw.d$ format treats the d specification as the *total* number of significant digits to print. If possible, the number is printed as a floating-point number and the place where the exponent goes is padded with blanks. Otherwise, the number is printed in exponential form.

The best way to see how the G format works is to print a column of numbers of various sizes. The numbers are placed in the field so that the numbers without an exponent line up under the numbers that do have an exponent. The following example compares the output produced by the F13.7, E13.7, and G13.7 formats. A field width of 13 was selected to represent the seven significant digits of the REAL*4 data, plus six overhead characters (sign, decimal point, E, sign, and two digits for the exponent). Because numbers are normalized for exponential format, seven digits after the decimal point must be specified to have all significant digits printed.

```

PROGRAM real_formats

* This program shows a comparison of the F, E, and G
* format descriptors:

REAL*4 realvalue

realvalue = 0.1234567E-3

WRITE(6, '(12X, "F13.7", 16X, "E13.7", 16X, "G13.7")')

DO i = 1, 14

    WRITE(6,100) realvalue, realvalue, realvalue

    realvalue = realvalue * 10.0

END DO
100 FORMAT (6X,"{",F13.7,"}",6X,"{",E13.7,"}",6X,"{",G13.7,"}")

END

```

The output of the above program follows:

F13.7	E13.7	G13.7
{ .0001234}	{ .1234567E-03}	{ .1234567E-03}
{ .0012345}	{ .1234567E-02}	{ .1234567E-02}
{ .0123456}	{ .1234567E-01}	{ .1234567E-01}
{ .1234567}	{ .1234567E+00}	{ .1234567 }
{ 1.2345669}	{ .1234567E+01}	{ 1.234567 }
{ 12.3456688}	{ .1234567E+02}	{ 12.34567 }
{ 123.4566956}	{ .1234567E+03}	{ 123.4567 }
{ 1234.5668945}	{ .1234567E+04}	{ 1234.567 }
{12345.6699219}	{ .1234567E+05}	{ 12345.67 }
{*****}	{ .1234567E+06}	{ 123456.7 }
{*****}	{ .1234567E+07}	{ 1234567. }
{*****}	{ .1234567E+08}	{ .1234567E+08}
{*****}	{ .1234567E+09}	{ .1234567E+09}
{*****}	{ .1234567E+10}	{ .1234567E+10}

Numbers are always right-justified into the output field, with the exception of non-exponential data formatted by the G descriptor, as shown above.

Note that in the list printed by the F descriptor, only the leftmost seven digits of the number are significant. Also, positive numbers over 99999.99 cannot be printed in F13.7 format because eight places are taken up by the seven fractional digits and the decimal point, leaving only five places for digits to the left of the decimal point. If the numbers are negative, the sign takes up another one of these places and the minimum negative number becomes -9999.999. When a number is out of range for the specified field, the field is filled with asterisks indicating a range of more than six orders of magnitude.

Character Format Descriptors: A, R

The A and R format descriptors define fields for character data. The forms of the descriptors are:

$$\begin{array}{l} A[w] \\ R[w] \end{array}$$

where w is the width of the field. If the field width w is omitted, the width of the field is equal to the length of the variable on input, or equal to the length of the character expression on output.

The purpose of format descriptors is to convert characters between their internal representation and a string of ASCII characters. Character data, however, is represented internally in ASCII format; therefore, the A format descriptor merely specifies a field for input or output characters.

If the A format descriptor is used without the w field width specification, the field width is automatically selected to be the same size as the character variable in the I/O list. For example, the program

```
PROGRAM char_data

CHARACTER*10 first, last

first = 'Jane'
last = 'Smith'

WRITE(6, '(1X, "My name is ", 2A)') first, last
WRITE(6, '(1X, "My name is ", 2A)') first(1:5), last(1:6)
END
```

writes the following:

```
My name is JaneΔΔΔΔΔΔSmithΔΔΔΔΔΔ
My name is JaneΔSmithΔ
```

The first WRITE statement effectively uses a 2A10 format descriptor because the variables `first` and `last` were declared to be 10 bytes in length. The second WRITE statement effectively uses A5, A6 format

descriptors because the substrings specified in the output variable list are 5 and 6 bytes in length, respectively.

The Rw format descriptor is an HP extension to the ANSI 77 Standard, which provides compatibility with other versions of FORTRAN. The difference between Aw and Rw occurs only when the specified field width w is *less than* the number of characters specified by the I/O variable. The differences are as follows:

- Using the Aw format descriptor, input data is left-justified into the input variable. The remaining positions are blank-filled. On output, the leftmost characters of the output variable are written. This is shown in Figure 2-1.

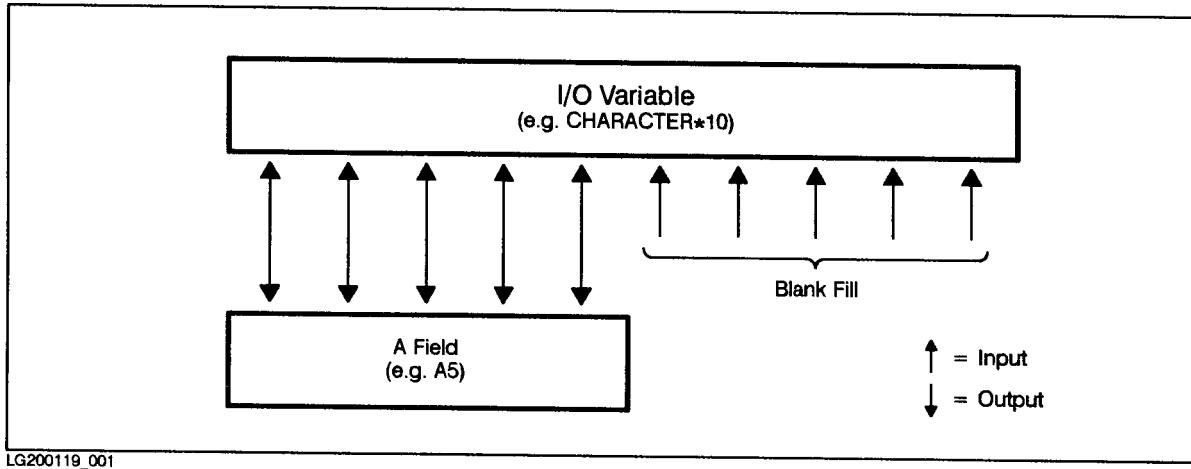


Figure 2-1. The Aw Format Descriptor

- Using the Rw format descriptor, input data is right-justified into the input variable. The initial positions are blank-filled. (ASCII null characters are represented by a byte of all blanks.) On output, the rightmost characters of the output variable are written. This is shown in Figure 2-2.

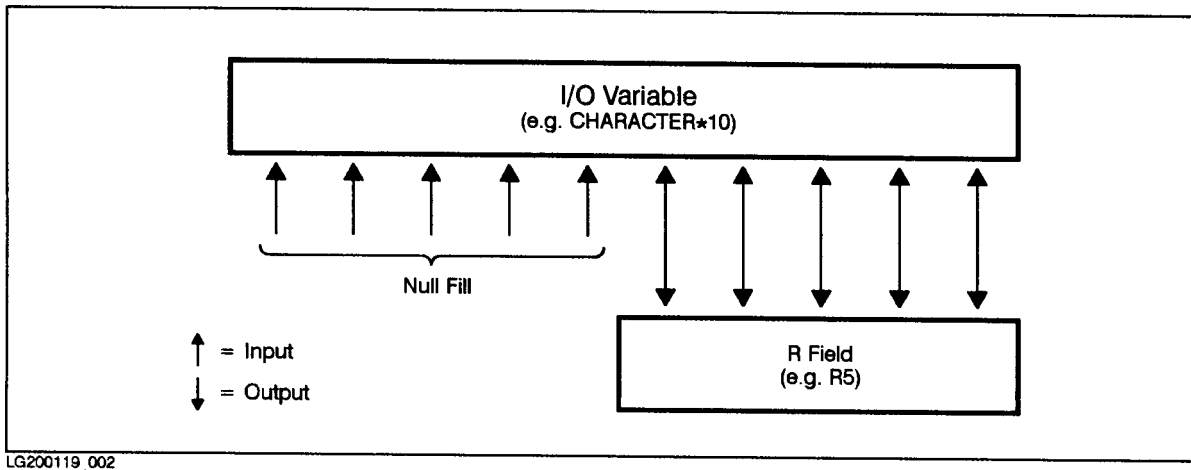


Figure 2-2. The Rw Format Descriptor

If the specified field width w is *greater than* the number of characters specified by the I/O variable, the Aw and Rw format descriptors behave in the same way. Input data is taken from the rightmost characters of the input field. Output data is right-justified into the output field. This is shown in Figure 2-3.

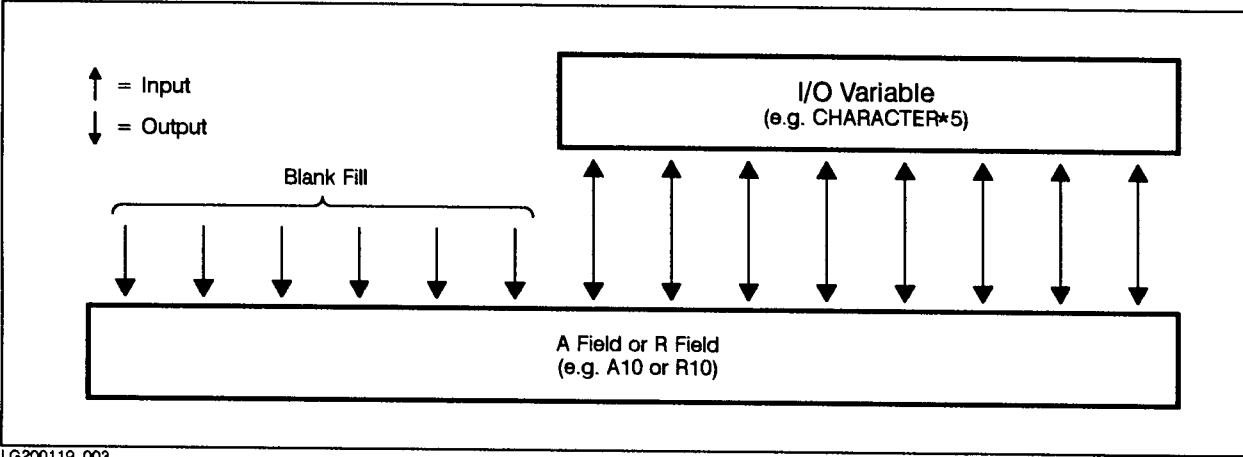


Figure 2-3. The Aw and Rw Format Descriptors

The Input Field

With the A descriptor, if the field width w is less than the length of the character variable, the characters are stored left-justified in the variable with the remainder of the variable filled with blank characters.

With the R descriptor, if the field width w is less than the length of the character variable, the characters are stored right-justified in the variable, preceded by null characters.

For example, the program:

```
PROGRAM widthsmaller_input

CHARACTER char1*10          ! Declare char1 to be 10 characters
CHARACTER char2*10          ! Declare char2 to be 10 characters

READ (5, '(A3)') char1     ! Read only 3 characters
READ (5, '(R3)') char2     ! Read only 3 characters

WRITE (6, '(1X, A)') char1
WRITE (6, '(1X, R)') char2

END
```

with the input:

```
ABC
ABC
```

writes the following:

```
ABC
ABC
```

Note that the null characters are not printable. The actual value stored in `char1` is:

```
ABCΔΔΔΔΔΔΔΔ
```

where Δ represents a blank space. The value stored in `char2` is:

```
□□□□□□□□ABC
```

where \square represents eight binary zeros, or the ASCII null character (the null character is equivalent to `CHAR (0)`).

With the A or R descriptor, if the field width w is larger than the length of the character variable, the rightmost characters are stored and the remaining characters are ignored.

For example, the program below shows how character data is input.

```
PROGRAM widthlarger_input

CHARACTER char1*5           ! Declare char1 to be 5 characters
CHARACTER char2*5           ! Declare char2 to be 5 characters

READ (5, '(A10)') char1    ! Read 10 characters
READ (5, '(R10)') char2    ! Read 10 characters

WRITE (6, '(1X, A)') char1 ! Print what was stored in char1
WRITE (6, '(1X, R)') char2 ! Print what was stored in char2

END
```

If the input to this program is:

```
ABCDEFGHIJ
ABCDEFGHIJ
```

the value stored in `char1` and `char2` is:

```
FGHIJ
```

The Output Field

With the A descriptor, if the field width w is less than the length of the character variable, the leftmost characters in the variable are output.

With the R descriptor, if the field width w is less than the length of the character variable, the rightmost characters in the variable are output.

For example, the program:

```
PROGRAM widthsmaller_output

CHARACTER char*10          ! Declare char to be 10 characters

char = 'ABCDEFGHIJ'

WRITE (6, '(1X, A3)') char ! Write only 3 characters
WRITE (6, '(1X, R3)') char ! Write only 3 characters

END
```

produces the following output:

```
ABC
HIJ
```

With the A or R descriptor, if the field width w is greater than the length of the character variable, the characters are right-justified in the field, with blanks on the left.

For example, the program:

```
PROGRAM widthgreater_output

CHARACTER char*5          ! Assign char to be 5 characters
char = 'ABCDE'

WRITE (6, '(1X, A10)') char ! Write 10 characters
WRITE (6, '(1X, R10)') char ! Write 10 characters

END
```

produces the following output:

```
ΔΔΔΔΔABCDE
ΔΔΔΔΔABCDE
```

The program below also shows how the A and R format descriptors differ. The program uses the input value `abcdef`.

```
PROGRAM char_ex

CHARACTER*3 alpha3, alpha3a
CHARACTER*9 alpha9, alpha9a

C INPUT using Aw and Rw format descriptors:

C Each READ statement gets this 6-character input field: abcdef

C   Input Statement:                Equivalent assignment:

      READ(5, '(A6)') alpha3         ! alpha3 = 'def'
      READ(5, '(R6)') alpha3         ! alpha3 = 'def'

      READ(5, '(A6)') alpha9         ! alpha9 = 'abcdefΔΔΔ'
      READ(5, '(R6)') alpha9         ! alpha9 = '###abcdef'
C                                     (^ represents a blank character)
C                                     (# represents a null character)

C OUTPUT using Aw and Rw format descriptors:

      alpha3a = 'abc'                 ! Assign data for
      alpha9a = 'abcdefghi'          ! output examples

C   Output Statement:                Characters written:

      WRITE(6, '(1X, A6)') alpha3a   ! ΔΔΔabc
      WRITE(6, '(1X, R6)') alpha3a   ! ΔΔΔabc
C                                     (Δ represents a blank character)

      WRITE(6, '(1X, A6)') alpha9a    ! abcdef
      WRITE(6, '(1X, R6)') alpha9a    ! defghi

      END
```

The A[w] and R[w] character format descriptors can be used with integer and real data types by specifying the NOSTANDARD compiler directive. The data is output in reverse order, starting at the right and progressing to the left.

For example, the following program:

```
program demo
c  Output numeric data with character format using an external write.

INTEGER*4 i4

i4 = 4Habcd
WRITE(6,100) i4
100 FORMAT(a)
STOP
END
```

produces the following output if \$NOSTANDARD or \$NOSTANDARD IO is specified:

dcba

or produces the following output if \$NOSTANDARD or \$NOSTANDARD IO is not specified:

abcd

For more information refer to the *HP FORTRAN 77/iX Reference*.

Logical Format Descriptor: L

The L format descriptor defines fields for logical data. The form of the descriptor is:

```
L[w]
```

where *w* is the width of the field.

The Input Field

If the first nonblank characters in the input field are T or .T, the value .TRUE. is stored in the logical variable. If the first nonblank characters in the input field are F or .F, the value .FALSE. is stored in the logical variable. If the first nonblank characters are not T, .T, F, or .F, an error occurs. Note that the lowercase letters t and f are also allowed.

For example, the program below reads logical values:

```
PROGRAM l_format_input
LOGICAL logical1, logical2

READ (5, '(L5)') logical1
READ (5, '(L2)') logical2

PRINT *, logical1, logical2

END
```

If the input to this program is:

```
ΔΔΔΔΔ
F1
```

the value stored in `logical1` is .TRUE. and the value in `logical2` is .FALSE..

The Output Field

The letter T or F is right-justified in the output field depending on whether the value of the list item is .TRUE. or .FALSE..

For example, the program:

```
PROGRAM l_format_output

LOGICAL logical1, logical2

logical1 = .FALSE.
logical2 = .TRUE.

WRITE (6, '(1X, L5)') logical1
WRITE (6, '(1X, L2)') logical2

END
```

produces the following output:

```
ΔΔΔΔF
ΔT
```

Repeating Specifications

The format descriptors can be repeated by prefixing the descriptor with a positive, unsigned integer specifying the number of repetitions.

For example, the statement:

```
100 FORMAT (3F10.5, 2I5)
```

is equivalent to:

```
100 FORMAT (F10.5, F10.5, F10.5, I5, I5)
```

A group of descriptors can be repeated by enclosing the group with parentheses and prefixing the group with a positive, unsigned integer.

For example, the statement:

```
100 FORMAT (I5, 3(F10.5,2X))
```

is equivalent to:

```
100 FORMAT (I5, F10.5, 2X, F10.5, 2X, F10.5, 2X)
```

Correspondence Between the I/O List and Format Descriptors

Usually there is a one-to-one correspondence between the items in the I/O list and the accompanying format descriptors. If, however, there are fewer items in the I/O list than corresponding format descriptors, the remaining format descriptors are ignored. For example, the statements:

```
a = 5.0
b = 7.0
WRITE(6, '(1X, F6.1, F6.2, " id = ", I5, I2)') a, b
```

do not use the I5 and the I2 descriptors. However, the "id=" character constant is printed as follows:

```
5.0 7.00 id=
```

The extra characters can be suppressed by using the colon edit descriptor. For example, the statement:

```
WRITE(6, '(1X, F6.1, F6.2, :, " id= ", I5, I2)') a, b
```

omits the trailing characters from the output line as follows:

```
5.0 7.00
```

See the colon edit descriptor description later in this chapter for further examples.

When there are more items in the output list than corresponding format descriptors, the FORMAT statement is reused from the beginning. If the FORMAT statement contains nested (parenthesized) format descriptors, reuse begins with the rightmost nested group at the first level. For example, the statements below show what portion of the FORMAT statement is reused.

```
WRITE(6,100) i, a, j, i1, a1, j1, i2, a2, j2
100 FORMAT(I3, 2X, F9.2, 2X, I5)
      ↑-----reused-----↑
```

```
WRITE(6, 200) i, a, j, a1, j1, a2, j2, a3, j3
200 FORMAT(I3, 2X, (F9.2, (2X, I5)))
      ↑---reused--↑
```

```
WRITE(6,300) i, a, j, j1, j2, j3, j4, j5, j6
300 FORMAT(I3, 2X, (F9.2, 2X), (I5))
      ↑↑
      reused
```

The format statements above treat the first three data items *i*, *a*, and *j* in each I/O list the same. However, the reused portion of the format specification is altered by nested format specifications. The first FORMAT statement shows that, in the absence of nested format descriptors, the entire list of format descriptors is reused. The second FORMAT statement shows how the reused portion of the FORMAT statement can contain additional nested specifications. The third FORMAT statement shows that not all nested format specifications are reused.

A program that does not have a one-to-one match between list elements and format descriptors is shown below:

```
PROGRAM unmatched

      READ(5, 100) a, i, a1, i1, a2, i2, a3
100  FORMAT(F4.1, (I5, F5.1))

      WRITE(6, 100) a, i, a1, i1, a2, i2, a3

      END
```

The program reads the variables as follows: *a* is input with format F4.1, *i* is input with format I5, and *a1* is input with format F5.1. Then, the number of format descriptors is exhausted. Flow control returns to the specification (I5, F5.1) and *i1* is input with format I5 and *a2* is input with format F5.1. Then, the number of format descriptors is again exhausted. As before, flow control returns to the specification (I5, F5.1) and *i2* is input with format I5 and *a3* is input with format F5.1.

Monetary Format Descriptor: M

The *Mw.d* format descriptor defines a field for a real number without an exponent (fixed-point) written in monetary form. The general specification is:

$$M[w.d]$$

where *w* is the width of the field and *d* is the number of digits after the decimal point.

The Input Field

On input, the *Mw.d* format descriptor causes interpretation of the next *w* positions of the input record as a real number without an exponent. The field width is expected (but not required) to have a dollar sign and comma(s) embedded in the data as described for *Mw.d* output (the dollar sign and commas are ignored). If commas are used, the usage must be consistent; that is, commas must occur every three digits of the nonfractional part of the input value. The field width can include “\$”, “&”, and “,”. The number is converted to an internal representation value for the variable (list element) currently using the format descriptor.

The input statement:

```
READ (5, '(M10.2)')a
```

can read any of the following input values:

Input	Assignment
123.45	a = 123.45
\$1234.56	a = 1234.56
\$1,234,567	a = 12345.67
\$12,345.4	a = 12345.40
1,234,567.99	a = 1234567.00
-1234.56	a = -1234.56
-\$123.75	a = -123.75
-\$1,357.91	a = -1357.91
1,234	a = 12.34
blanks	a = .00

The Output Field

On output, the $Mw.d$ format descriptor causes output of a numeric value in ASCII character fixed-point form, right-justified with commas and a dollar sign. The least significant digit (position d) is rounded. If needed, a leading minus sign is printed before the dollar sign.

In addition to the number of numeric digits, the field width w must allow for the number of commas expected plus four characters to hold the sign, the dollar sign, the decimal point, and a rollover digit (if necessary). If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required, the output value of the entire field is filled with asterisks.

```
PROGRAM m_format

C   This program demonstrates output with the monetary format

REAL*4 money
money = 12345.67

WRITE (6, '(25X, "M17.2")')

DO i = 1,14
    WRITE (6,100) money
    money = money * 3
END DO

100 FORMAT (18X, "{", M17.2, "}")
STOP
END
```

The above program produces the following output:

```
      M17 2
{      $12,345.67}
{      $37,037.01}
{     $111,111.02}
{     $333,333.06}
{     $999,999.19}
{    $2,999,997.50}
{    $8,999,992.00}
{   $26,999,976.00}
{   $80,999,928.00}
{  $242,999,776.00}
{  $728,999,296.00}
{ $2,186,997,760.00}
{ $6,560,993,280.00}
{*****}
```

**Numeration Format
Descriptor: N**

The *Nw.d* field descriptor defines a field for a real number without an exponent (fixed-point) written in numeration form (that is, with commas, which are then ignored, in the input field).

The general specification is:

$$N[w.d]$$

where *w* is the width of the field and *d* is the number of digits after the decimal point.

The Input Field

On input, the *Nw.d* field descriptor causes interpretation of the next *w* positions of the input record as a real number without an exponent. The field width is expected (but not required) to have commas embedded in the data as described for *Nw.d* output (the commas are ignored). If commas are used, the usage must be consistent; that is, commas must occur every three digits of the nonfractional part of the input value. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

The input statement:

```
READ (5, '(N10.2)')
```

can read any of the following input values:

Input	Assignment
123.56	a = 123.56
12,345.66	a = 12345.66
1,224,666	a = 12246.66
-13,555.87	a = -13555.87
+5,987.54	a = 5987.54
1,234,567.88	a = 1234567.00
3,456.78	a = 3456.78
4,567.89	a = 4567.89
blanks	a = .00

The Output Field

On output, the *Nw.d* field descriptor causes output of a numeric value in ASCII character fixed-point form, right-justified with commas. The least significant digit is rounded. If needed, a leading minus sign is printed before the most significant digit.

In addition to the number of numeric digits, the field width *w* must allow for the number of commas expected, plus three characters to hold the sign, the decimal point, and a rollover digit (if necessary). If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the output value, the entire field is filled with asterisks.

```
PROGRAM n_format

C   This program demonstrates output with the numeric format

REAL*4 num
num = 12345.67

WRITE (6, '(25X, "N17.2")')

DO i = 1,14
  WRITE (6,100) num
  num = num * 3
END DO

100 FORMAT (18X, "{", N17.2, "}")
STOP
END
```

The above program produces the following output:

```
      N17.2
{      12,345.67}
{      37,037.01}
{     111,111.02}
{     333,333.06}
{     999,999.19}
{    2,999,997.50}
{    8,999,992.00}
{   26,999,976.00}
{   80,999,928.00}
{  242,999,776.00}
{  728,999,296.00}
{ 2,186,997,760.00}
{ 6,560,993,280.00}
{19,682,979,840.00}
```

Processing New Lines

The /, NN, NL, and \$ descriptors handle the control of new lines.

The / Descriptor

The / edit descriptor terminates the current line and begins processing a new input or output line. On input, a slash indicates that data will come from the next line; on output, a slash indicates that data will be written to the next line.

Commas separating edit descriptors and separating consecutive slashes are not needed.

For example, the following program uses the / descriptor for input and output:

```
PROGRAM slash_edit

  READ (5, 100) inta, intb, reala
100  FORMAT (I5, I3/F5.3)

  WRITE (6, 200) inta, intb, reala
200  FORMAT
*(1X, 'Integer values = ',I5,' and ',I3 // ' Real value = ',F5.3)

END
```

If the input to this program is:

```
12345123
1.234
```

the output looks like this:

```
Integer values = 12345 and 123

Real value = 1.234
```

The NL, NN, or \$ Descriptor

The NL, NN, or \$ edit descriptor controls the carriage return at the end of an output line or record. For compatibility with other versions of FORTRAN, the \$ edit descriptor is equivalent to the NN descriptor. For a detailed description, refer to the *HP FORTRAN 77/iX Reference*.

Handling Character Positions

The X, T, TL, and TR edit descriptors handle character position control.

The X Descriptor

The X edit descriptor skips character positions in an input or output line. The form of the descriptor is:

$$nX$$

where n is the number of positions to be skipped from the current position; n must be a positive nonzero integer.

On input, the X edit descriptor causes the next n positions of the input line to be skipped. On output, the X descriptor causes n positions of the output line to be filled with blanks, if not previously defined; these positions are not otherwise written. The X descriptor is identical to the TR descriptor.

For example, the following program uses the X descriptor for input and output.

```
PROGRAM x_edit

INTEGER a, b

READ (5, 100) a, b, c
100 FORMAT (2X, I3, 5X, I3, F9.4)

WRITE (6,200) a, b, c
200 FORMAT (5X, I3, 5X, I3, 5X, F9.4)

END
```

If the input to this program is:

```
ΔΔ123ΔΔΔΔΔ1231234.5678
```

the program produces this output:

```
ΔΔΔΔΔ123ΔΔΔΔΔ123ΔΔΔΔΔ1234.5678
```

The T Descriptor

The T edit descriptor provides tab control. The form of the descriptor is:

$$Tn$$

where n is a positive, nonzero integer indicating number of columns.

When the T edit descriptor is on a format line, input or output control skips right or left to the character position n ; the next descriptor is then processed. Be careful not to skip beyond the length of the record.

For example, consider this program:

```
PROGRAM t_edit

      READ (5, 100) a, b
100  FORMAT (T6, F4.1, T15, F6.2)

      PRINT 200, a, b
200  FORMAT (F4.1, 5X, F6.2)

      END
```

If the input to this program is:

```
ΔΔΔΔΔ12.3ΔΔΔΔΔ123.45
      ↑           ↑
      Column     Column
      6           15
```

the value stored in `a` is 12.3 and the value stored in `b` is 123.45.

Using the T edit descriptor, you can write over fields; that is, you can destroy a previously formed field. For example, the program:

```
PROGRAM t_edit_2
      OPEN (3, FILE='writeit')
      WRITE (3, 100)
100  FORMAT (1X, '1234567890', T4, 'abcde')
      CLOSE (3)
      END
```

writes the following string to file `writeit`:

```
12abcde890
```

Similarly, you can also reread fields with the T descriptor.

The TL Descriptor

The TL edit descriptor provides tab control. The form of the descriptor is:

`TLn`

where n is a positive, nonzero integer indicating number of columns.

When the TL edit descriptor is on a format line, input or output control skips left n column positions from the current cursor position. If n is greater or equal to the current cursor position, the control goes to the first column position.

For example, consider this program:

```
PROGRAM tl_edit

      OPEN (3, FILE='datafile')
      READ (3, 100) a, b
100   FORMAT (F6.2, TL6, F6.2)
      PRINT 200, a, b
200   FORMAT (1X, F6.2, 5X, F6.2)

      CLOSE (3)
      END
```

If the file `datafile` contains the data:

```
123.45
```

the output looks like this:

```
123.45 123.45
```

Using the TL edit descriptor, you can write over fields. For example, the program

```
PROGRAM tl_edit

      OPEN (3, FILE='writeit')
      WRITE(3, 100)
100   FORMAT(1X, 'It is winter ', TL7, 'summer.')
```

```
      CLOSE(3)
      END
```

writes the line:

```
It is summer.
```

to the file `writeit`.

The TR Descriptor

The TR edit descriptor provides tab control. The form of the descriptor is:

`TR n`

where n is a positive, nonzero integer indicating number of columns.

When the TR edit descriptor is on a format line, input or output control skips right n column positions from the current cursor position. Be careful not to skip beyond the length of the record.

For example, the program:

```
PROGRAM tr_edit

a = 123.4
b = 1234.11

WRITE (6, 100) a, b
100 FORMAT (1X, F5.1, TR5, F7.2)

END
```

produces the following output:

```
123.4ΔΔΔΔΔ1234.11
```


Handling Literal Data The ', ", and H edit descriptors handle literal data.

The ' and " Descriptors

Paired apostrophe (') and quotation mark (") edit descriptors write character strings; the paired symbols delimit a string of characters, which can include blanks. The ' and " descriptors are preferred over the H edit descriptor.

If the character string contains an apostrophe or a quotation mark, you can do one of the following:

- Delimit the symbol with two marks of the same type
- Use the other symbol as the delimiter

For example, the program:

```
PROGRAM literal_edit

WRITE (6,100)      ! String uses apostrophes
100  FORMAT (1X, 'Enter your name:')

WRITE (6,200)      ! String uses quotation marks
200  FORMAT (1X, "Enter your address:")

WRITE (6,300)      ! Statement is continued on two lines
300  FORMAT (1X, 'Enter your employee number',
*      ' and employee location code:')

WRITE (6,400)      ! Quotation mark with two marks of the same type
400  FORMAT (1X, 'What''s your home telephone number?')

WRITE (6,500)      ! Quotation mark with other symbol as delimiter
500  FORMAT (1X, "What's your work telephone number?")

END
```

produces the following output:

```
Enter your name:
Enter your address:
Enter your employee number and employee location code:
What's your home telephone number?
What's your work telephone number?
```

The H Descriptor

The H (Hollerith) edit descriptor writes character strings. The H descriptor has the form:

nHstring

where *n* is the number of characters in the string and *string* is the string of characters. The string of characters is not delimited with quotation marks.

For example, the program:

```
PROGRAM h_edit
  pi = 3.14159
  WRITE (6, 100) pi
100  FORMAT (1X, 21HThe value of "pi" is , F7.5)
END
```

produces the following output:

```
The value of "pi" is 3.14159
```

The H descriptor is provided for compatibility with older versions of FORTRAN; its use is discouraged.

Using Scale Factors: The P Descriptor

The P edit descriptor scales real numbers on input or output. The descriptor has the form:

nP

where *n* is the integer scale factor. The P descriptor can precede the D, E, and G format descriptors for input and output without an intervening comma or other separator.

Once a P descriptor is specified, the scale factor holds for all subsequent descriptors on the FORMAT statement until another scale factor is defined. A scale factor of zero (0P) ends the effect of the scale factor.

On input, the scale factor affects fixed-field values; the value is multiplied by 10 raised to the *-nth* power. However, if the input number includes an exponent, the scale factor has no effect.

For example, this program shows how the P descriptor affects numeric values:

```
PROGRAM p_edit_input

READ (5, '(G8.4)') value1      ! Multiply by 10**0

READ (5, '(-2PG8.4)') value2  ! Multiply by 10**(2)

READ (5, '(2PG8.4)') value3   ! Multiply by 10**(-2)

READ (5, '(2PG8.4)') value4   ! If input includes an exponent,
*                               the scale factor has no effect.
```

END

If the input to this program is:

```
123.4567
123.4567
123.4567
123.45E0
```

the values stored are as follows:

```
value 1 = 123.4567
value 2 = 12345.67
value 3 = 1.234567
value 4 = 123.4500
```

On output, the scale factor affects the D, E, and F format descriptors. The scale factor affects the G format descriptor only if *Gw.d* is interpreted as *Ew.d*.

When using the P edit descriptor with the D, E, or G format descriptor, the forms are as follows:

nPD or *nPDw.d*

nPE or *nPEw.d* or *nPEw.dEe*

nPG or *nPGw.d* or *nPGw.dEe*

n is a required value. If you do not specify *n*, the default value will be 1. *w.d* and *Ee* are both optional, but if you specify *Ee*, you must specify *w.d*

When using the P edit descriptor with the G format descriptor, the input or output is dependent on the value to be read or written. The scale factor *nP* shifts the decimal point to the right *n* places and reduces the exponent by *n*.

When using the P edit descriptor with the F format descriptor, the form is as follows:

nPFw.d

The internal value is multiplied by 10^n .

For example, the program:

```
PROGRAM p_edit_output

pi = 3.14159

C Write without a scale factor
WRITE (6, '(2X, "FORMAT", 10X, "VALUE"/)')
WRITE (6, '(1X, " D10.4", 5X, D10.4/)') pi

C Write with a scale factor on the D and E format descriptors
WRITE (6, '(1X, "-3PD10.4", 5X, -3PD10.4)') pi
WRITE (6, '(1X, "-1PE10.4", 5X, -1PE10.4)') pi
WRITE (6, '(1X, " 1PE10.4", 5X, 1PE10.4)') pi
WRITE (6, '(1X, " 3PD10.4", 5X, 3PD10.4)') pi
WRITE (6, '(1X, " 3PE10.4", 5X, 3PE10.4/)') pi

C Write with a scale factor on the F format descriptor
WRITE (6, '(1X, "-1PF10.4", 5X, -1PF10.4)') pi
WRITE (6, '(1X, " PF10.4", 5X, PF10.4)') pi
WRITE (6, '(1X, " 5PF10.4", 5X, 5PF10.4)') pi

END
```

produces the following output:

FORMAT	VALUE
D10.4	.3142D+01
-3PD10.4	.0003D+04
-1PE10.4	.0314E+02
1PE10.4	3.1416E+00
3PD10.4	314.16E-02
3PE10.4	314.16E-02
-1PF10.4	.3142
PF10.4	31.4159
5PF10.4	*****

Note



The last field is filled with asterisks because the field width w is smaller than the number of digits needed to represent the value.

If the P descriptor does not precede a D, E, F, or G descriptor, it should be separated from other descriptors by commas or slashes. For example, the statement:

```
100 FORMAT(1X, 2P, 3(I5, F7.2))
```

scales the F format descriptor value.

If the P descriptor does precede a D, E, F, or G descriptor, the comma or slash is optional.

For example, the program:

```
PROGRAM p_edit_output_2

int = 5
real = 2.2
pi = 3.14159

* Output values without a scale factor:
WRITE (6,50) int, real, pi
50 FORMAT (1X, I2, 3X, F14.4, 3X, E15.4/)
* Output values with a scale factor:
WRITE (6,100) int, real, pi
100 FORMAT (1X, I2, 3X, 3P, F14.4, 3X, E15.4)
* Show that FORMAT statements 100, 200, and 300 are equivalent:
WRITE (6,200) int, real, pi
200 FORMAT (1X, 3P, I2, 3X, F14.4, 3X, E15.4)
WRITE (6,300) int, real, pi
300 FORMAT (1X, I2, 3X, 3PF14.4, 3X, E15.4)

END
```

produces the following output:

```
5          2.2000          .3142E+01

5      2200.0000          314.16E-02
5      2200.0000          314.16E-02
5      2200.0000          314.16E-02
```

Note that the P descriptor has no effect on the I2 format descriptor.

Printing Plus Signs: The S, SP, and SS Descriptors

The S, SP and SS edit descriptors can be used with the D, E, F, G, and I format descriptors to control the printing of optional plus signs (+) in numeric output. A formatted output statement does not usually print the plus signs. However, if an SP edit descriptor is in a format specification, all succeeding positive numeric fields will have a plus sign. The field width w must be large enough to contain the sign. When an S or SS edit descriptor is encountered, the optional plus signs are not printed.

For example, the program:

```
PROGRAM s_edit

int = 12345

WRITE(6,100) int           ! By default, + is not printed
100 FORMAT(1X, I5)

WRITE(6,200) int, int      ! With SP, the + is printed
200 FORMAT(1X, SP, I6, /, 1X, S, I6) ! With S, the + is not printed
END
```

produces the following output:

```
12345
+12345
12345
```

Returning the Number of Bytes: The Q Descriptor

The Q edit descriptor returns the number of bytes remaining on the current input record. The value is returned to the next item on the input list, which must be an integer variable. This descriptor applies to input only and is ignored by the WRITE or PRINT statements.

For example, in the program:

```
PROGRAM q_format_input

CHARACTER string(80)

READ(5,100) len, (string(i), i = 1, min(len, 80))
100 FORMAT(Q, 80A1)

END
```

the variable `len` gets assigned the current length of the string. The Q edit descriptor is used to avoid an error from reading more bytes from the input record than are available, or from having blanks provided that were not in the input file.

Terminating Format Control: The Colon Descriptor

The colon (:) edit descriptor conditionally terminates format control, just as if the final right parenthesis in the FORMAT statement has been reached. If there are more items in the I/O list, the colon edit descriptor has no effect.

For example, the program:

```
PROGRAM colon_edit

value1 = 12.12
value2 = 34.34
value3 = 56.56

WRITE(6,100) value1, value2, value3
100 FORMAT(1X, 'Values = ', 3(F5.2, :, ', '))

END
```

produces the following output:

```
Values = 12.12, 34.34, 56.56
```

The format control terminated after the value of `value3` was printed, not after the final comma was printed.

The colon has no effect on input except if a / descriptor is included. For example, if the contents of file `datafile` are:

```
12
24
36
48
```

the program:

```
PROGRAM example1

OPEN(9, FILE='datafile')
READ (9,10) i
READ (9,10) j
100 FORMAT (5(I2, /))
PRINT *, i
PRINT *, j

CLOSE(9)
END
```

produces the following output:

```
12
36
```


However, the program:

```
PROGRAM example2

OPEN(9, FILE='datafile')
READ (9,10) i
READ (9,10) j
10  FORMAT (5(I2, :, /))
PRINT *, i
PRINT *, j

CLOSE(9)
END
```

produces this output:

```
12
24
```

In the first example, the / descriptor causes the record containing the value 24 to be skipped. In the second example, the colon terminates format control before the / descriptor because no more list items remain in the I/O list.

Handling Blanks in the Input Field

The BN and BZ edit descriptors are used to handle blanks in the input field.

The BN Descriptor

The BN edit descriptor is used with the D, E, F, G, I, O, K, @, and Z format descriptors to interpret blanks in numeric input fields. If BN is specified, all embedded blanks are ignored, the input number is right-justified within the field width, and, if needed, the field is padded with leading blanks. An input field of all blanks has a value of zero. If the BN or BZ descriptors are not specified, the treatment of blanks is as if you specified the BN edit descriptor. An exception to this default is when the unit is connected with BLANK='ZERO' specified in the OPEN statement, as described in Chapter 3.

For example, consider this program:

```
PROGRAM bn_edit

      READ (5, 100) int1, val1, int2
100  FORMAT (I3, BN, F6.2, I3)

      END
```

If the input to this program is:

```
1ΔΔΔ4Δ.Δ2Δ1Δ
```

the variables `int`, `val1`, and `int2` have the following values:

```
int1 = 1
val1 = 4.2
int2 = 1
```

The BN edit descriptor remains in effect until a BZ edit descriptor (described below) is encountered or until the end of the format specification.

For example, the following program uses the BZ descriptor to cancel the BN descriptor's treatment of blanks.

```
PROGRAM bn_bz_edit

      READ (5, 100) int1, int2, int3
100  FORMAT (I5, BN, I3, BZ, I5)

      PRINT *, int1, int2, int3

      END
```

If the input line is:

```
1ΔΔΔΔ2ΔΔ3ΔΔΔΔ
```

the variables `int1`, `int2`, and `int3` have the following values:

```
int1 = 1
```

```
int2 = 2  
int3 = 30000
```

The BZ Descriptor

The BZ edit descriptor is used with the D, E, F, G, I, O, K, @, and Z format descriptors to interpret blanks in numeric input fields. If BZ is specified, trailing and embedded blanks are interpreted as zeros. An input field of all blanks has a value of zero.

For example, consider this program:

```
PROGRAM bz_edit

      READ (5,100) int1, val1, int2
100  FORMAT (I3, BZ, F6.2, I3)

      END
```

If the input to this program is:

```
1ΔΔΔ4Δ.Δ2Δ1Δ
```

the variables `int`, `val1`, and `int2` have the following values:

```
int1 = 1
val1 = 40.02
int2 = 10
```

The BZ edit descriptor remains in effect until a BN edit descriptor is encountered or until the end of the format specification.

For example, the program below uses the BN descriptor to end the interpretation of blanks as zeros.

```
PROGRAM bn_bz_edit

      READ (5,100) int1, int2, int3
100  FORMAT (I5, BZ, I3, BN, I5)

      PRINT *, int1, int2, int3

      END
```

If the input line is:

```
1ΔΔΔΔ2ΔΔ3ΔΔΔΔ
```

the variables `int1`, `int2`, and `int3` have the following values:

```
int1 = 1
int2 = 200
int3 = 3
```

Alternative Methods of Specifying Input/Output

There are alternative methods of specifying an I/O statement. They are:

- Using the PARAMETER statement to assign input and output unit numbers
- Using character variables to represent formats
- Using the ASSIGN statement to assign a FORMAT label to an INTEGER*4 variable

These methods are shown in the following program:

```
PROGRAM formatting

CHARACTER*16  format_string
INTEGER*4     format_label

C Using the PARAMETER statement:
  INTEGER*4 in, out
  PARAMETER (in=5, out=6)
  READ(in,*) a, b, c
  WRITE(out,*) a, b, c

C Using CHARACTER variables to represent formats:
  format_string = '(1X, I9)'
  READ(5, format_string) i
  format_string = '(1X, "i=", I9)'
  WRITE(6, format_string) i

C Using the ASSIGN statement:
  ASSIGN 100 TO format_label      ! Note: format_label must be a
  READ(5, format_label) i        ! 4-byte integer
  ASSIGN 200 TO format_label
  WRITE(6, format_label) i
100 FORMAT(I9)
200 FORMAT('1X, i=', I9)

END
```

Using the Implied DO Loop

The implied DO loop is used with the READ, WRITE, and PRINT statements. An implied DO loop contains a list of data elements to be read or written, and a set of indexing parameters. Following is an implied DO loop:

```
PRINT *, (apple, i = 1,3)
```

where `apple` is the index parameter and `i` is the data element.

The statement above prints the value of `apple` three times. If `apple` is initialized to 35.6, the output would look like this:

```
35.6 35.6 35.6
```

If the list of an implied DO loop contains several variables, each of the variables in the list is input or output for each pass through the loop. For example, the statement:

```
READ *, (a,b,c, j = 1,2)
```

is equivalent to the list-directed statement:

```
READ *, a, b, c, a, b, c
```

An implied DO loop is often used to input or output arrays and array elements. For example, the statements:

```
READ b(10)
PRINT *, (b(i), i=1,10)
```

result in the array `b` written in the following order:

```
b(1) b(2) b(3) b(4) b(5) b(6) b(7) b(8) b(9) b(10)
```

If an unsubscripted array name is used in this list, the entire array is transmitted. For example, the statements:

```
READ x(3)
PRINT *, (x, i= 1,2)
```

write the elements of array `x` two times as follows:

```
x(1) x(2) x(3) x(1) x(2) x(3)
```

On output, the list can contain expressions that use the index value. For example, the statements:

```
READ a(10)
PRINT *, (i*2, a(i*2), i= 1,5)
```

write the numbers 2, 4, 6, 8, 10, alternating with array elements `a(2)`, `a(4)`, `a(6)`, `a(8)`, `a(10)`.

Implied DO loops are useful for controlling the order in which arrays are output. You can output an array in column-major or row-major order. Suppose you have the following program:

```
PROGRAM implieddo
INTEGER a1(2,3)
DATA a1 /1, 2, 3, 4, 5, 6/
WRITE (6, '(1X, 3I2)') a1
WRITE (6, '(1X, 3I2)') ((a1(i,j), j = 1,3), i=1,2)
END
```

The statement:

```
WRITE (6, '(1X, 3I2)') a1
```

writes the array elements in column-major order, like this:

```
1 2 3
4 5 6
```

The statement:

```
WRITE (6, '(1X, 3I2)') ((a1(i,j), j = 1,3), i=1,2)
```

writes the array in row-major order, like this:

```
1 3 5
2 4 6
```

Because FORTRAN stores arrays in column-major order, these two statements produce the same result:

```
WRITE (6, '(1X, 3I2)') ((array(i,j),i = 1,2), j = 1,3)
```

```
WRITE (6, '(1X, 3I2)') array
```

The following program initializes a 10 by 10-element array as an identity matrix. An identity matrix has a diagonal of ones and the rest of the array is filled with zeros. The program uses a WRITE statement with an implied DO loop to output the array in row-major order.

```
PROGRAM array

INTEGER id_array(10,10)
DATA ((id_array(i,j), j = i+1,10), i=1,9) /45*0/ ! upper
DATA (id_array(i,i), i=1,10) /10*1/ ! diagonal
DATA (id_array(i,j), i = j+1,10), j=1,9) /45*0/ ! lower
WRITE(6,'(1X, 10I2)') ((id_array(i,j), j = 1,10), i=1,10)

END
```

The program produces this output:

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

Implied DO loops for input or output are not just used with arrays. The following program prints a table of degrees and the sine of each, in steps of 10 degrees.

```
PROGRAM sine

WRITE (6,100) (d, SIN(d*3.14159/180.), d=0,360,10)
100 FORMAT (1X, F4.0, F9.5)

END
```


The program produces the following output:

0.	.00000
10.	.17365
20.	.34202
30.	.50000
40.	.64279
50.	.76604
60.	.86602
70.	.93969
80.	.98481
90.	1.00000
100.	.98481
110.	.93969
120.	.86603
130.	.76605
140.	.64279
150.	.50000
160.	.34202
170.	.17365
180.	.00000
190.	-.17365
200.	-.34202
210.	-.50000
220.	-.64279
230.	-.76604
240.	-.86602
250.	-.93969
260.	-.98481
270.	-1.00000
280.	-.98481
290.	-.93969
300.	-.86603
310.	-.76605
320.	-.64279
330.	-.50000
340.	-.34202
350.	-.17365
360.	-.00001

File Handling

HP FORTRAN 77 performs input/output operations with a wide range of devices, including disk drives, terminals, and line printers, as well as the computer's own memory. Internal file I/O provides a way to perform data conversions with character data in computer memory. This chapter describes I/O operations with disk files and internal files.

Disk Files

The source or destination of an HP FORTRAN 77 I/O operation is specified by a unit number. The preconnected units 5 and 6 were used in Chapter 2 for I/O with the standard input and output devices. To access data on a disk file, you must first connect the file to a unit number with the OPEN statement. For example, the statement:

```
OPEN(9, FILE='payroll')
```

connects unit number 9 with the disk file `payroll`. The OPEN statement sets the file pointer to point to the first record in the file. The file `payroll` can now be read or written with I/O statements specifying unit 9, as follows:

```
WRITE(9, '(3I5)') i, j, k
```

or:

```
READ(9, '(3I5)') i, j, k
```

READ or WRITE statements access the current record in the file; the current record is pointed to by the file pointer. After the I/O operation, the file pointer automatically moves to the next record in the file, making that record the new current record.

Normally an I/O statement reads or writes one record of a file. In fact, a record can be thought of as the data read or written by a single I/O statement. Later, however, we will see that it is possible to access more than one record in a file with a single READ or WRITE statement.

When a program has finished the file I/O activity, the unit number should be “disconnected” from the file with the CLOSE statement. The statement:

```
CLOSE(9)
```

breaks the connection between unit 9 and the connected file. The CLOSE statement has the effect of flushing buffers used for file I/O and releasing the unit number for connection to another file if necessary. When a program terminates, an automatic CLOSE is performed on each connected unit in the program. However, it is good practice to include a CLOSE statement when file I/O is complete.

Default File Properties

The statements shown so far illustrate the simplest file handling commands possible. The file `payroll` created is the default file type for HP FORTRAN 77; that is, the file is a sequential access, formatted file with unknown status. These three major properties of HP FORTRAN 77 files are described below.

ACCESS='SEQUENTIAL'

If ACCESS is not specified in the OPEN statement, the file will be a sequential access file. An important characteristic of sequential access files is that each record in the file can be a different number of bytes in length. Consequently, records in a sequential file must be read or written in sequential order. This is easy to do because the file pointer is set to the first record when the file is opened and advances automatically with each I/O statement. Later in this chapter, you will see how to open a direct access file.

FORM='FORMATTED'

If FORM is not specified in the OPEN statement of a sequential access file, the file will be a formatted file. A formatted file has data in the form of ASCII characters. I/O statements that access formatted files must use format specifications or list-directed I/O. The formatter converts the data between its internal form (in computer memory) to its ASCII representation (in the file). Later in this chapter, you will see how to open an unformatted file.

STATUS='UNKNOWN'

If STATUS is not specified in the OPEN statement, the file will have an unknown status. The status of a file refers to whether the file exists when the OPEN statement executes. A file with unknown status will be created if it does not already exist before being connected to the unit number in the OPEN statement. Later in this chapter, you will see how to open a file with new, old, or scratch status.

Reporting File Handling Errors

In addition to connecting a unit number to a file, the OPEN statement also flags certain file errors by using the STATUS specifier, transfers control when an error occurs by using the ERR specifier, and returns the error message number by using the IOSTAT specifier.

The STATUS Specifier

The STATUS specifier in the OPEN statement assigns a status to the file. The four types of status are described in Table 3-1.

Table 3-1. Status Types

Status Type	Description
'NEW'	The file will be created by the OPEN statement. If the file exists, an error occurs. This status is useful to protect against writing over an existing file.
'OLD'	The file is expected to already exist. If the file does not exist, an error occurs. This status is useful for programs that use existing data files.
'UNKNOWN'	If the file does not exist, the file is created; otherwise, the existing file is used. This is the default file status.
'SCRATCH'	A file is created, named by FORTRAN, and deleted when the file is closed or when the program ends. Scratch files do not use the FILE='filename' specifier. This status is useful for programs that need a temporary file.

For example, the statement:

```
OPEN(2, FILE='myfile', STATUS='NEW')
```

opens the file `myfile` with new status.

The STATUS specifier in the CLOSE statement assigns a closing status to the file. The two types of status are described in Table 3-2.

Table 3-2. The STATUS Specifier

Status Type	Description
'KEEP'	The file will be saved on the disk. This is the default file status.
'DELETE'	The file will be purged from the disk.

For example, the statement:

```
CLOSE(2, FILE='myfile', STATUS='KEEP')
```

closes the file `myfile` with keep status, so the file will be saved on the disk.

A scratch file is always deleted by the system at the end of the session. If you specify the CLOSE statement with STATUS='KEEP', an error occurs.

The ERR Specifier

The ERR specifier in the OPEN statement assigns a statement label for the program to jump to when an error occurs. For example, the statement:

```
OPEN(9, FILE='datafile', STATUS='NEW', ERR=180)
```

connects the logical unit 9 to the file `datafile`. If an error occurs in the opening of the file, control transfers to the statement labeled 180.

The IOSTAT Specifier

The IOSTAT specifier in the OPEN statement names the integer variable where the system returns the error message number. The values returned in this integer variable are summarized in Table 3-3.

Table 3-3. The IOSTAT Specifier

Value of IOSTAT Integer Variable	Meaning
Zero	No error occurred.
Greater than zero	An error occurred; an error message number is returned.

The error message number refers to runtime errors. Refer to the *HP FORTRAN 77/iX Reference* for a description of the error message. The statement:

```
OPEN(4, FILE='xyzfile', ERR=99, IOSTAT=ios)
```

connects the logical unit number 4 to the file `xyzfile`. If an error occurs in the opening of the file, the error number is placed in the variable `ios` and control transfers to the statement labeled 99.

The CLOSE statement also reports file handling errors. For example, the statement:

```
CLOSE(16, IOSTAT=ios, ERR=99, STATUS='DELETE')
```

disconnects the file that was connected to unit number 16 and specifies that the file should be deleted. If an error occurs, control transfers to the statement labeled 99 and the error number is stored in the variable `ios`.

Creating a New File Using STATUS='NEW'

The OPEN statement with STATUS='NEW' creates a new file. A file can be created by one program and used by another.

The program below creates a file and uses the OPEN statement specifiers STATUS, ERR, and IOSTAT to leave the existing file unchanged with repeated runs of the program.

```
PROGRAM open_specifiers1

INTEGER*4  account_num, number, quantity
REAL*4    price

C Create file "pfile".
C If an error occurs, ios will contain the runtime error number.
C Because STATUS='NEW' is specified, an error will occur if
C "pfile" already exists.

      OPEN(9, FILE='pfile', STATUS='NEW', ERR=999, IOSTAT=ios)

C Prompt for data and read data from the standard I/O device:

      WRITE(6, '(1X, "Enter number of products: ",NN)')
      READ(5, *)number

C Write to the file "pfile":
      WRITE(9, '(1X, I10)') number

      DO i = 1, number
        WRITE(6, '(1X, "Enter quantity and price: ",NN)')
        READ(5,*) quantity, price
        WRITE(9, '(1X, I10, F9.2)') quantity, price
      END DO

C Close the file "pfile"; terminate connection to unit 9:
      CLOSE(9)
      STOP 'Normal termination'

999 CALL report_error(ios, 'OPEN', 'pfile')
      STOP "Error termination"
      END

SUBROUTINE report_error(ios, stmt_name, file_name)

INTEGER*4  ios, eof, found, not_found
CHARACTER  stmt_name*(*), file_name*(*)
PARAMETER (eof = -1, found = 918, not_found = 908)

      IF (ios .EQ. found) THEN
        WRITE(6,*) stmt_name, ' error: the file ',
*               file_name, ' already exists.'
      ELSE IF (ios .EQ. not_found) THEN
```



```

        WRITE(6,*) stmt_name, ' error: the file ',
*           file_name, ' was not found.'
ELSE IF (ios .LE. eof) THEN
    WRITE(6,*) stmt_name, ' End of file ', file_name
ELSE
    WRITE(6,*) stmt_name, ' error', ios, ' on file ', file_name
END IF

END

```

In this program, unit 9 is connected to the file `pfile`. You are prompted to enter data for the variables `number`, `quantity`, and `price`. The variable `number` represents the number of transactions in the `pfile` and controls the number of iterations through the `DO` loop. The `WRITE` statement to unit 9 writes the data to the file `pfile` with the format `(I10)` for `number` and the format `(I10, F9.2)` for `quantity` and `price`.

The error handling subroutine `report_error` prints a message depending on the error number assigned to `ios`.

In the program, the error handling subroutine uses these three variables:

Variable	Purpose
<code>ios</code>	Contains the I/O error number.
<code>stmt_name</code>	Contains the statement that caused the error.
<code>file_name</code>	Contains the file name that caused the error.

The error subroutine in this program includes checks for some specific errors. The error “File not found” (reported for STATUS='OLD' files) and “File already exists” (reported for STATUS='NEW' files) generate positive return values for ios. After the error routine, the STOP statement halts the program and prints the message “Error termination”.

A sample session of the program follows:

```
Enter number of products: 5
Enter quantity and price: 3, 5.16
Enter quantity and price: 2, 9.25
Enter quantity and price: 6, 1.72
Enter quantity and price: 1, 15.91
Enter quantity and price: 14, 2.75
STOP Normal Termination
```

After the program executes, the contents of the file pfile is:

```
5
3      5.16
2      9.25
6      1.72
1      15.91
14     2.75
```

If the file pfile already existed, an error occurs.

Reading From an Existing File Using STATUS='OLD'

To read data from an existing file, you must first open the file with the OPEN statement. For example, the following program opens and reads the existing file `pfile` that was created in the previous example. By specifying `STATUS='OLD'`, the file `pfile` must exist or else the program will terminate.

```
PROGRAM open_specifiers2

INTEGER*4 number, quantity
REAL*4    price
CHARACTER stmt_name*8

C Open file "pfile"; if it does not exist, go to statement 999:
  stmt_name = 'OPEN'
  OPEN(9, FILE='pfile', STATUS='OLD', ERR=999, IOSTAT=ios)

C Read the file to get the number of records;
C if an error occurs, go to statement 999:
  stmt_name = 'READ 1'
  READ(9, '(I10)', ERR=999, IOSTAT=ios) number

C Read the file to get the data;
C if an error occurs, go to statement 999:
  stmt_name = 'READ 2'
  DO i = 1, number
    READ(9, '(I10, F9.2)', ERR=999, IOSTAT=ios)
    *           quantity, price
    WRITE(6, '(1X, I5, I10, F9.2)') i, quantity, price
  END DO

  CLOSE(9)
  STOP 'Normal termination'

999 CALL report_error(ios, stmt_name, 'pfile')
  STOP 'Error termination'
  END
```

```

SUBROUTINE report_error(ios, stmt_name, file_name)

INTEGER*4 ios, eof, found, not_found
CHARACTER stmt_name*(*), file_name*(*)
PARAMETER (eof = -1, found = 918, not_found = 908)

IF (ios .EQ. found) THEN
  WRITE(6,*) stmt_name, ' error: the file ',
*           file_name, ' already exists.'
ELSE IF (ios .EQ. not_found) THEN
  WRITE(6,*) stmt_name, ' error: the file ',
*           file_name, ' was not found.'
ELSE IF (ios .LE. eof) THEN
  WRITE(6,*) stmt_name, ' End of file ', file_name
ELSE
  WRITE(6,*) stmt_name, ' error', ios, ' on file', file_name
END IF

END

```

This program reads the quantity and price from the file and prints each line. The variable `number` controls the loop that reads the records. Again, the `CLOSE` statement disconnects the unit from the file name.

One line of data is read or written by one `READ` or `WRITE` statement. The format descriptors specified in the `READ` or `WRITE` statement define the data of each line in the file. Therefore, each line must be read with the same format descriptors used to write the file.

When this program executes, the following is output:

```

      1          3      5.16
      2          2      9.25
      3          6      1.72
      4          1     15.91
      5         14      2.75
      STOP Normal termination

```

If the file `pfile` did not exist, an error occurs.

Appending to a File

To write data to an existing file, you must first open the file with the OPEN statement. For example, the following program opens and writes to the existing file `prices`, whose contents are as follows:

```
3      5.16
2      9.25
6      1.72
1      15.91
14     2.75
```

The program first finds the end of the file, and then accepts the new data.

```
PROGRAM write_exist

LOGICAL  forever
PARAMETER (forever = .TRUE.)
INTEGER*4 number, quantity, count
REAL*4   price

C  Open the file "prices" and connect it to unit 8:
   OPEN(8, FILE = 'prices', STATUS='OLD')

C  Position the file pointer to the end of the file:
   DO WHILE (forever)
     READ(8, '(X)', END=100)
   END DO

C  Backspace to write over the end-of-file record
100  BACKSPACE 8

C  Get new data and write the data to the file "prices":
   WRITE(6,*) 'How many records do you want to add?'
   READ(5,*) number

   DO i = 1, number
     WRITE(6,*) 'Enter quantity and price: '
     READ(5,*) quantity, price
     WRITE(8, '(1X, I10, F9.2)') quantity, price
   END DO

C  Close the file; terminate connection to unit 8:
   CLOSE(8)

END
```

A sample run is shown below:

```
How many records do you want to add? 2
Enter quantity and price: 1, 1.50
Enter quantity and price: 5, 2.25
```

After the program executes, the contents of `prices` look like this:

```
3      5.16
2      9.25
6      1.72
1     15.91
14     2.75
1      1.50
5      2.25
```

The new data is appended to the end of the existing file.

File Access

The examples so far have been sequential access files; unless otherwise specified in the `OPEN` statement, files are opened for sequential access. But, FORTRAN files can be accessed (read or written) in three ways: sequential, direct, or indexed sequential.

Sequential Access Files

Sequential access files are read and written in sequence; that is, files are accessed in the order in which they were written. In a sequential access file, the following is true:

- The file pointer points to the current record.
- The `READ` or `WRITE` statement operates on the current record.
- After the `READ` or `WRITE`, the file pointer advances to the next record.

The end of a sequential file is marked by an end-of-file (EOF) record. A diagram of how a sequential access file is structured is shown in Figure 3-1.

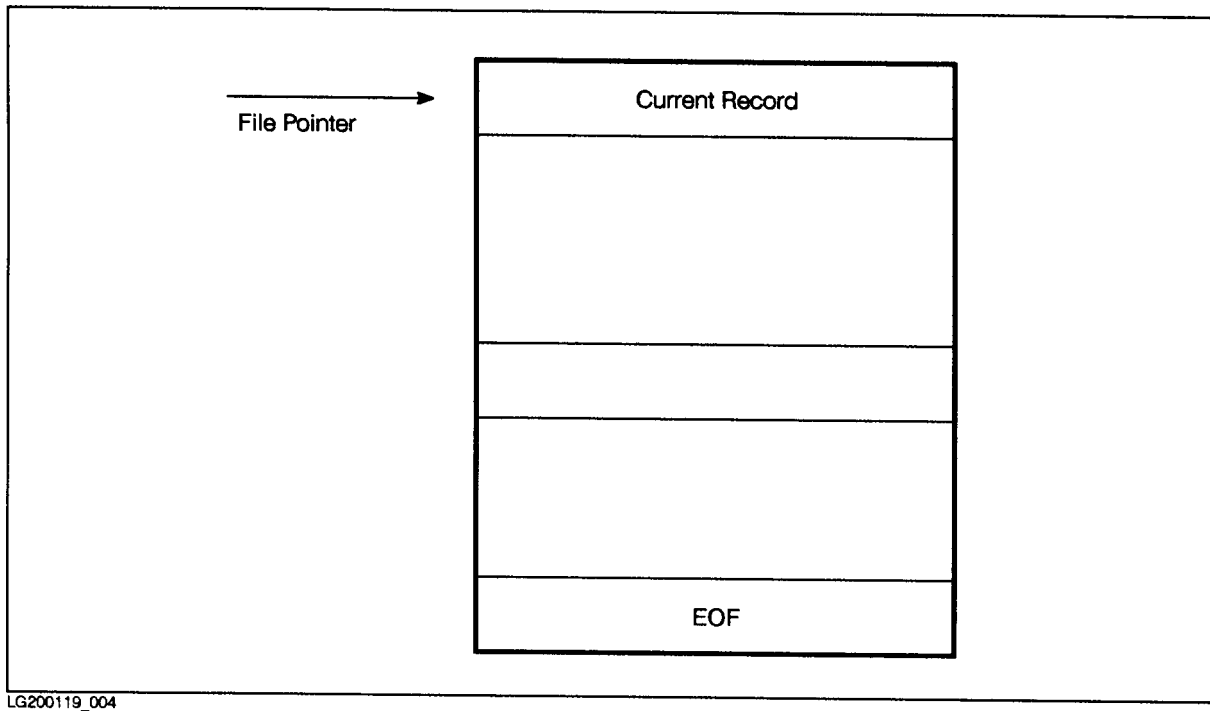


Figure 3-1. Sequential Access Files

Note that each record in a sequential access file can be a different size.

The default specifiers in the OPEN statement for sequential access files are summarized below:

Specifier	Default
STATUS	'UNKNOWN'
ACCESS	'SEQUENTIAL'
FORM	'FORMATTED'
BLANK	'NULL'

The BLANK specifier describes how blanks within numbers are treated on input from formatted files. If `BLANK='NULL'`, blanks are ignored; if `BLANK='ZERO'`, blanks are treated as zeros.

For example, the statement: `OPEN(1, FILE='infile', STATUS='OLD', BLANK='ZERO')` connects a file named `infile` to logical unit number one. The file `infile` exists as a sequential file for formatted I/O. All blanks will be treated as zeros on input.

Direct Access Files

Direct access files are read and written according to the record number; the record number can then be used in random order. Each record in the file has a record number that is specified by the REC specifier on a READ or WRITE statement. Each record in a direct file must be the same size, as specified in the OPEN statement. A diagram of how a direct access file is structured is shown in Figure 3-2.

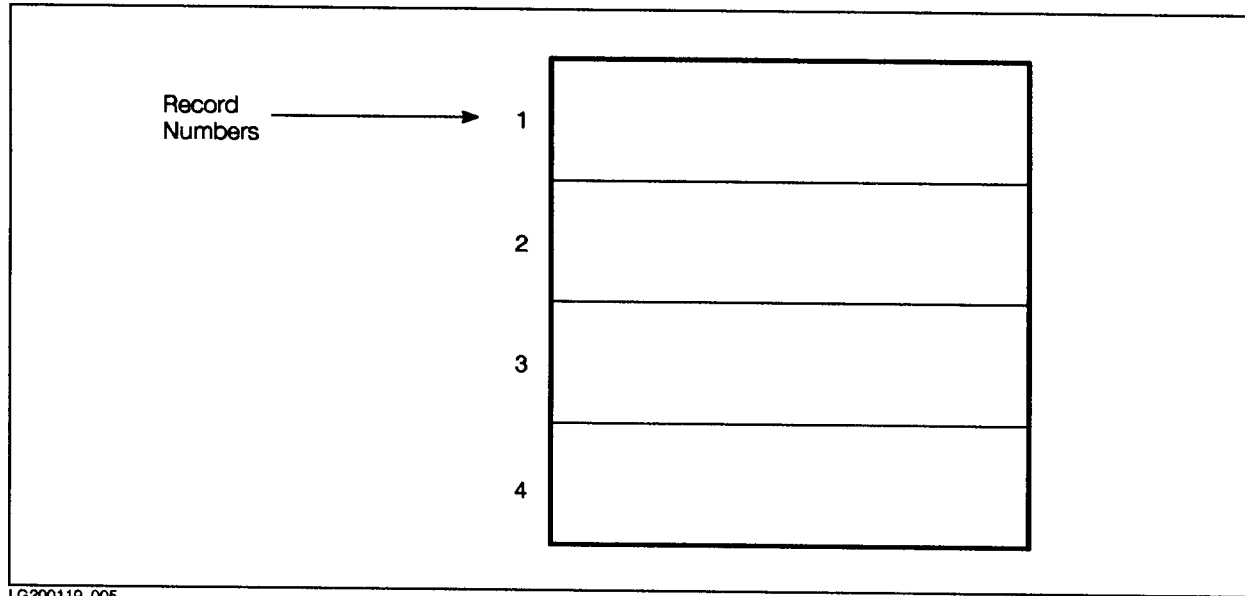


Figure 3-2. Direct Access Files

Note that each record is the same size and that there is no end-of-file (EOF) record in a direct access file.

Once established, the record number of a specific record cannot be changed or deleted, although the record can be rewritten. The records can be read or written in any order. For example, record number 3 can be written before writing record number 1.

The records of a direct access file cannot be read or written using list-directed formatting. Because a direct access file does not have an end-of-file record, the END specifier in the direct access READ or WRITE statement is not allowed.

The default specifiers in the OPEN statement for direct access files are summarized below:

Specifier	Default
STATUS	'UNKNOWN'
ACCESS	'DIRECT'
FORM	'UNFORMATTED'
BLANK	'NULL'

If FORM='FORMATTED' is specified, BLANK defaults to 'NULL'.

The file format of a direct access file can be formatted or unformatted, as described below:

Type of Format	Description
FORMATTED	Records are blank-filled to the specified record length.
UNFORMATTED	Records are null-filled to the specified record length.

To create a direct access file, specify an OPEN statement with the ACCESS='DIRECT' specifier and the RECL (record length) specifier. For example, the statement:

```
OPEN(2, FILE='dfile', ACCESS='DIRECT', RECL=120)
```

opens the file `dfile` for direct access. The file is associated with unit two and has a record length of 120 bytes.

A temporary scratch file can be a direct access file. The statement:

```
OPEN(4, STATUS='SCRATCH', ACCESS='DIRECT', RECL=120)
```

connects a direct access scratch file to the FORTRAN unit four.

The program below shows how to create and write data to a direct access file.

```
PROGRAM direct_access

INTEGER quantity

C Open the file "dfile" for direct access
C (the record length is 120 bytes and the default type is unformatted):

OPEN(2, FILE='dfile', ACCESS='DIRECT', RECL=120)

C Prompt for number of transactions:
WRITE(6, '(A,NN)') "Enter number of transactions:"
READ(5,*) number

C Enter data and write the data to the direct access file using
C the id number as the record number:

DO i = 1, number
  WRITE(6, '(A,NN)') "Enter id number: "
  READ(5,*) id

  WRITE(6, '(A,NN)') "Enter quantity and price: "
  READ(5,*) quantity, price

  WRITE(2, REC=id) quantity, price
END DO

CLOSE(2)

END
```

A sample run of the program is shown below:

```
Enter number of transactions: 5
Enter id number: 4
Enter quantity and price: 14 16.00
Enter id number: 1
Enter quantity and price: 9 9.25
Enter id number: 6
Enter quantity and price: 1 142.90
Enter id number: 2
Enter quantity and price: 60 1.50
Enter id number: 5
Enter quantity and price: 3 74.70
```

When the program executes, the records 4, 1, 6, 2, and 5 are written to the file dfile.

The program below reads and prints the contents of the first five records in reverse order from the file `dfile`.

```
PROGRAM direct_read

INTEGER quantity

C Open the file "dfile"

OPEN(2, FILE='dfile', ACCESS='DIRECT', RECL=120)

C Read and print the first five records:

DO i = 5, 1, -1
  READ(2, REC=i) quantity, price
  WRITE(6, '(1X, I5, I5, F9.2)') i, quantity, price
END DO

END
```

The program produces the following output:

5	3	74.70
4	14	16.00
3	0	0.00
2	60	1.50
1	9	9.25

Note that record 3 is filled with zeros because that record is empty.

The sequential operations `READ`, `WRITE`, `BACKSPACE`, `ENDFILE`, and `REWIND` can be used on direct access files.

Indexed Sequential Access Files

Indexed sequential files (ISAM) are read and written randomly by a key or sequentially without a key. The key is part of the record. Files are of fixed length. Each record in the file has a primary key and one or more secondary or optional keys. These keys are part of the record that is being written into the file. Each record in an ISAM file must be the same size as specified in the OPEN statement by RECL. The ACCESS specifier in an OPEN statement should be specified as 'KEYED' for ISAM. An ISAM file has two physical files; a data file containing user records and an index file holding indexes to the records in the data file.

A diagram of how an indexed sequential file is structured is shown in Figure 3-3.

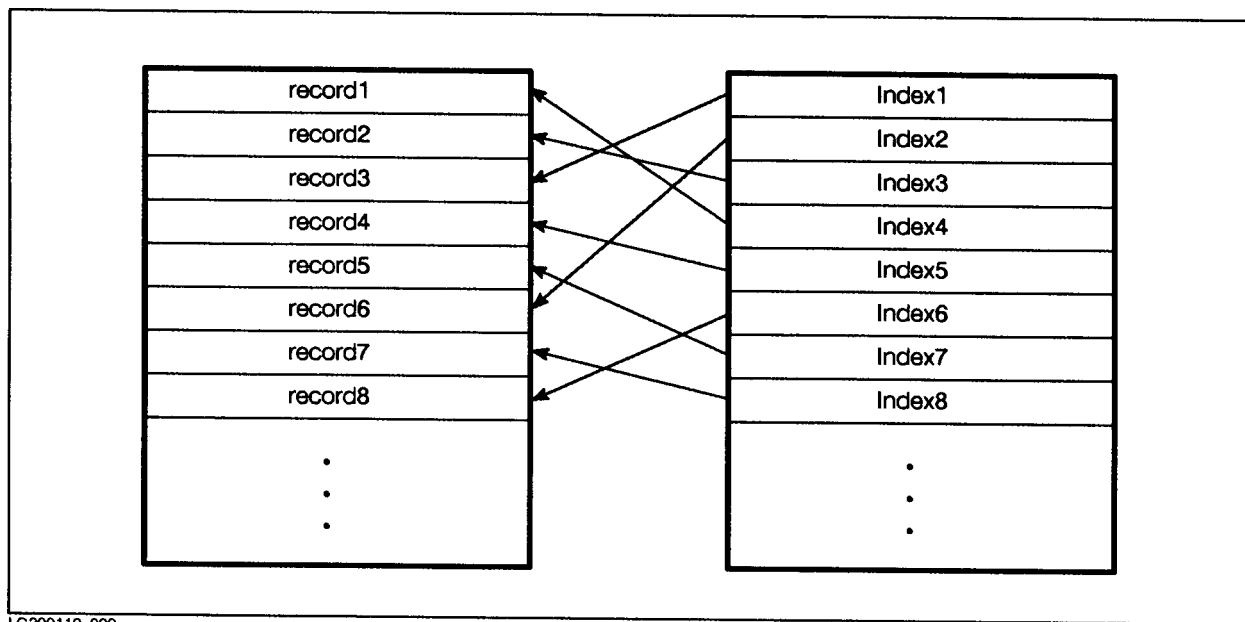


Figure 3-3. Indexed Sequential Access Files

The system will create two files one with the name given in the OPEN statement and the other with a K appended to the file name. If the original file name in the OPEN statement is already eight characters long, the system will replace the last character of the file name with a K. For example, a file named DATAFILE would have K appended to it as follows: DATAFILK.

The following table shows how a file is specified variable or fixed length in an OPEN statement.

ACCESS	RECL	RECORDTYPE	File Type.
SEQUENTIAL	Absent		Variable file.
SEQUENTIAL	Present		Variable file (RECL= maximum record length)
DIRECT	Absent		Error.
DIRECT	Present		Fixed record length file.
KEYED	Present	Variable	Error.
KEYED	Present	Fixed	Fixed record length ISAM file.
KEYED	Absent	Fixed	Error.

For example, the statement:

```
OPEN (10,FILE='file1',KEY=(1:4:integer,10:15:character),ACCESS='keyed')
```

connects a file named file1 to logical unit 10.

The program below shows how to create and write data to an indexed sequential file.

```
PROGRAM isam_write

STRUCTURE /cust/
  INTEGER*4    phone_num      ! primary key
  CHARACTER*15 last_name      ! first alternate key
  CHARACTER*15 first_name
  CHARACTER*1  middle_init
  CHARACTER*20 street
  CHARACTER*15 city
  CHARACTER*2  state
  INTEGER*4    zip
  CHARACTER*6  %fill
END STRUCTURE
RECORD /cust/ customer_rec
CHARACTER*1    temp

C   Open a file "test" for indexed (keyed) access with record length
C   of 80 bytes and with phone number as the primary key, and the
C   last name of the customer as the secondary key.

OPEN (10,FILE='test',ACCESS='keyed',RECL=80,FORM='unformatted',
1     RECORDTYPE='fixed',KEY=(1:4:INTEGER,5:19:CHARACTER))

8  WRITE (6,'(A,NN)') "continue to add(Y/N)? "
   READ (5,'(A1)') temp
   IF (temp .eq. 'N') GOTO 100

WRITE (6,'(A,NN)') "phone number: "
READ (5,*) customer_rec.phone

WRITE (6,'(A,NN)') "last_name: "
READ (5,10) customer_rec.last_name

WRITE (6,'(A,NN)') "first name: "
READ (5,10) customer_rec.first_name

WRITE (6,'(A,NN)') "middle init: "
READ (5,11) customer_rec.middle_init

WRITE (6,'(A,NN)') "street: "
READ (5,12) customer_rec.street

WRITE (6,'(A,NN)') "city: "
```

```

READ (5,10) customer_rec.city

WRITE (6,'(A,NN)') "state: "
READ (5,13) customer_rec.state

WRITE (6,'(A,NN)') "zip: "
READ (5,*) customer_rec.zip

WRITE (10,err=200) customer_rec
GOTO 8

10 FORMAT(A15)
11 FORMAT(A1)
12 FORMAT(A20)
13 FORMAT(A2)

100 STOP
200 PRINT *,' error in writing'
END

```

The program below shows how to read an ISAM file with an alternate key and to read it sequentially.

```

PROGRAM isam_read
STRUCTURE /cust/
    INTEGER*4    phone_num      ! primary key
    CHARACTER*15 last_name     ! first alternate key
    CHARACTER*15 first_name
    CHARACTER*1  middle_init
    CHARACTER*20 street
    CHARACTER*15 city
    CHARACTER*2  state
    INTEGER*4    zip
    CHARACTER*6  %fill
END STRUCTURE

CHARACTER*15    temp_name
LOGICAL         done

C   Open the isam file created in the previous example.
C   key, recl, and recordtype are optional for an existing file

OPEN (10,FILE='test',FORM='unformatted',STATUS='old',RECL=80,
1     RECORDTYPE='fixed',KEY=(1:4:INTEGER,5:19:CHARACTER))

```

```

C   Read a record with an alternate key( last name ) and list all the
C   names with that name as the last name

      WRITE(6,'(A,NN)') " enter the last name: "
      READ (5,10) temp_name
10  FORMAT (A15)

      READ (10,KEYEQ=temp_name,KEYID=1,ERR=101) customer_rec

      WRITE (6,11) customer_rec.phone_num, customer_rec.last_name,
1   customer_rec.first_name
11  FORMAT (I7,X4,A,X2,A)

C   Read sequentially and list phone number and the name until
C   last name changes.

      done = .false.
      DO WHILE (not done)
          READ (10,KEYEQ=temp_name,KEYID=1,ERR=101) customer_rec
          IF (customer_rec.last_name = temp_name) THEN
              WRITE (6,11) customer_rec.phone_num, customer_rec.last_name,
1   customer_rec.first_name
          ELSE
              done = .true.
          ENDIF
      ENDDO
      STOP
101 PRINT *, 'error in reading'
      END

```


Unformatted I/O

Unformatted I/O statements do not use format descriptors and do not convert the data on input or output. The data is transferred in internal (binary) representation to the external device. Unformatted I/O cannot be used for internal or formatted files. Note that terminals are opened as formatted files.

Unformatted Input

The unformatted READ statement transfers one line from the specified unit to the storage locations of the variables listed in the READ statement list.

For example, the statement:

```
READ (5) i, flag, ready
```

places values directly into `i`, `flag`, and `ready` without any data type conversions from character to internal (binary) form.

The data type of each input value should agree with the type of the corresponding list item.

The following program shows the unformatted input statements.

```
PROGRAM unformatted_input
INTEGER i(1000)
OPEN(9, FILE = 'bdata', FORM = 'UNFORMATTED')
```

```
C Unformatted input statements from unit 9.
C (The array i is read in internal binary format from unit 9.)
```

```
READ(9) i

CLOSE(9)

END
```

Unformatted Output

The unformatted WRITE statement writes to the specified output unit without any format conversion.

For example, the statement:

```
WRITE (6) i, flag, ready
```

writes the values of `i`, `flag`, and `ready` without any format conversion.

If a list is omitted in the WRITE statement, a null line is written.

Using Formatted and Unformatted Files

FORTRAN files can be either formatted or unformatted. If not specified in the OPEN statement, sequential files are defaulted to be formatted and direct access files are defaulted to be unformatted.

A formatted file is read and written with formatted I/O statements. That is, the READ or WRITE statements contain a format or list-directed specification.

An unformatted file is read and written with unformatted I/O statements. That is, the READ and WRITE statements do *not* contain format specifications.

For example, in the program below, unit 10 is connected to a formatted file and unit 11 is connected to an unformatted file. Both files are sequential access files.

```
PROGRAM format_unformat

REAL*4      value
INTEGER*4   count
CHARACTER*16 name

DATA value /123.456/
DATA count /123456/
DATA name  /'John Doe'/

C  Open the formatted and unformatted sequential files:

OPEN(10, FILE='file1', FORM='FORMATTED')
OPEN(11, FILE='file2', FORM='UNFORMATTED')

C  Write the data to the formatted file
C  (36 bytes will be written):

WRITE(10, '(F10.3, I10, A16)') value, count, name

C  Write the 24 data bytes to the unformatted file
C  (24 data bytes will be written):

WRITE(11) value, count, name

CLOSE(10)
CLOSE(11)

END
```

The formatted WRITE statement uses the (F10.3, I10, A16) format specification, which specifies that 36 characters are to be written to the disk.

The unformatted WRITE statement writes the data to the file without conversion. The number of bytes written to the disk correspond to the number of bytes allocated to the variables in the declaration statement. In this example, there is a 4-byte real, a 4-byte integer, and a 16-byte character variable, for a total of 24 bytes. The advantage of the unformatted WRITE statement is that the numeric data takes up less room on the disk and that the slow conversion from binary to character format is not done. For floating point data, using unformatted I/O ensures that accuracy is not lost by the conversion and rounding process used for ASCII I/O.

Using the INQUIRE Statement

The INQUIRE statement returns information about a file or unit. The information returned can be about one of the following:

- A file that is not connected to a unit
- A file that is connected to a unit
- A FORTRAN I/O unit

For example, the statement:

```
INQUIRE(FILE='abcfile', ERR=999, EXIST=ex, ACCESS=ac)
```

returns information about the file `abcfile` to the variables `ex` and `ac`. If an error occurs during the INQUIRE, control transfers to the statement labeled 999.

Here is another INQUIRE statement:

```
INQUIRE(FILE='exfile', IOSTAT=ios, ERR=99, EXIST=ex,  
*        OPENED=iop, NUMBER=num, ACCESS=acc)
```

This statement requests information on the file `exfile`. If `exfile` exists and is connected to a unit in the program, the variables `ex` and `iop` return the value `true`. The unit number of the file is stored in `num` and the character variable `acc` is defined.

If the file `exfile` does not exist, `ex` and `iop` return the value `false`, and `ios` is zero if there was no error, or is a system-defined value greater than zero if there was an error.

The program below opens the file `pfile` and uses the `INQUIRE` statement to return information about the file to the variables `ios`, `iop`, `ex`, and `ac`.

```
PROGRAM inquire_info

LOGICAL ex, iop
CHARACTER*10 ac

OPEN(8, FILE='pfile')

INQUIRE(FILE='pfile', IOSTAT=ios, OPENED=iop,
*        EXIST=ex, ACCESS=ac, ERR=100)

PRINT *, "iop = ", iop    ! iop returns T if the file is opened
PRINT *, "ex = ", ex     ! ex returns T if the file exists
PRINT *, "ac = ", ac     ! ac returns the type of file access

IF (ios .NE. 0) THEN     ! ios returns the I/O status
    PRINT *, "ios =", ios
ENDIF

100 CLOSE(8)

END
```

Because the file `pfile` exists and successfully opens, the program prints the following:

```
    iop = T
    ex = T
    ac = SEQUENTIAL
```

Similarly, the program below uses the INQUIRE statement to return information about unit 8. Note that this INQUIRE statement refers to the connection to I/O unit 8, not to the file pfile.

```
PROGRAM inquire_info2

LOGICAL      ex, iop
CHARACTER*10 ac

OPEN(8, FILE='pfile')

INQUIRE(UNIT=8, IOSTAT=ios, OPENED=iop,
*        EXIST=ex, ACCESS=ac, ERR=100)

PRINT *, ios ! ios returns the I/O status
PRINT *, iop ! iop returns T if the unit is opened
PRINT *, ex  ! ex returns T if the unit exists
PRINT *, ac  ! ac returns the type of file access

100  CLOSE(8)

END
```

Because unit 8 is connected, the program prints the following:

```
0
T
T
SEQUENTIAL
```

Positioning the File Pointer

The `BACKSPACE`, `REWIND`, and `ENDFILE` statements control the position of the file pointer within a sequential access file.

The unit specifiers `UNIT`, `IOSTAT`, and `ERR` can be used on the file positioning statements.

The `BACKSPACE` Statement

The `BACKSPACE` statement positions the file pointer back one record. For example, the statement:

```
BACKSPACE 10
```

moves the file pointer for unit 10 back one record.

The `REWIND` Statement

The `REWIND` statement positions the file pointer at the beginning of the file. For example, the statement:

```
REWIND(UNIT=13, IOSTAT=ios, ERR=99)
```

moves the file pointer to the initial point in the file connected to the logical unit 13. If an error occurs, the error number is stored in the variable `ios` and control transfers to statement 99.

The `ENDFILE` Statement

The `ENDFILE` statement writes an end-of-file record as the next record. For example, the statement:

```
ENDFILE 13
```

writes an end-of-file record as the next record of the file connected to unit number 13.

Example of Using the File Positioning Statements

The file positioning statements are used in the following program:

```
PROGRAM positioning

INTEGER*4 quantity
REAL*4    price

C Open the file connected to unit 10:

    OPEN(10, FILE = 'pfile')

C Read some records:

    DO i = 1, 5
        READ(10, '(I10, F9.2)') quantity, price
    END DO

C Move the file pointer from unit 10 to the previous record:

    BACKSPACE 10

C Move the file pointer to the initial point in the file connected
C to logical unit 10.
C If an error occurs, the error number is
C stored in the variable ios and control transfers to statement 99.

    REWIND(10, IOSTAT=ios, ERR=99)

C Write an end-of-file record as the next record of the file connected
C to unit number 10:

    ENDFILE 10

    STOP

C Error handling section:

99  WRITE(6, '("ERROR = ", I6)') ios

    END
```


File Handling Examples

This section demonstrates the use of several options of the file handling statements.

Computing the Mean of Data in a Sequential File

The following program computes the mean of all the data items in the disk file `data`. The file contains an unknown number of records, with each record containing one real number.

```
PROGRAM compute_mean

sum = 0.0      ! Initialize
n = 0         ! Initialize
OPEN(3, IOSTAT = ios, ERR = 99, FILE = 'data',
*   ACCESS = 'SEQUENTIAL', STATUS = 'OLD')

10  READ (3, 22, END=88, IOSTAT = ios, ERR = 99) anum
22  FORMAT (F10.5)
    sum = sum + anum !Add data entries
    n = n + 1       ! Count entries
    GO TO 10        ! Loop

C Out of loop
88  avg = sum / n
    WRITE(6,33) avg ! Output to preconnected unit 6
33  FORMAT (1X, 'The average is ', F12.6)
    CLOSE(3)
    STOP

C   If there is an error in the OPEN or READ, output to the
C   preconnected unit 6.
99  WRITE(6,44) ios
44  FORMAT (1X, 'Error encountered = ', I6)

END
```

If the file `data` contains the following:

```
1.0
2.0
3.0
4.0
```

the output of the program looks like this:

```
The average is 2.500000
```

If the file `data` does not exist, an error occurs and the error number is output.

Inserting Data Into a Sorted Sequential File

This program inserts a single number in the proper position in a sorted sequential file.

```
PROGRAM insert_number

C Declare and initialize variables:
  IMPLICIT NONE
  REAL      anum, fnum
  INTEGER   ios1, ios2, EOF
  PARAMETER (EOF = -1)

C Open the files:
  OPEN(18, FILE='oldfile', STATUS='UNKNOWN', IOSTAT=ios1, ERR=99)
  OPEN(17, FILE='newfile', STATUS='NEW', IOSTAT =ios2, ERR=99)

C Prompt for number to be inserted and begin reading the file:
  PRINT *, 'Enter number to be inserted: '
  READ *, anum
  READ(18, *, END=100, IOSTAT=ios1, ERR=99) fnum

C Copy fnum to 'newfile' until EOF is reached or until fnum ≥ anum
  DO WHILE (fnum .LT. anum)
    WRITE(17, *) fnum
    READ(18), *, END=100, ERR=99, IOSTAT=ios1) fnum
  END DO

C Write the inserted number to 'newfile':
100 WRITE(17, *) anum

C Copy data to 'newfile' until EOF is reached:
  DO WHILE(ios1 .NE. EOF)
    WRITE(17, *) fnum
    READ(18, *, ERR=99, IOSTAT=ios1) fnum
  END DO

  CLOSE(17)
  CLOSE(18)

  STOP 'All Done'

C Error handling section
99  WRITE (6, '(1X, "ERROR = ", 2I6)') ios1, ios2

END
```

If the file `oldfile` originally contained the following:

```
1.0
2.0
3.0
```

4.0
5.0

and you run the program to insert the number 3.5, the file `newfile` looks like this after execution:

1.0
2.0
3.0
3.5
4.0
5.0

Internal Files

Internal files provide a way of reformatting, converting, and transferring data from one area of memory to another; no input or output devices are used. If you use internal files to reformat data, you do not have to write data to a file and reread the file with a different format. Instead, internal files allow conversion between numeric and character data types.

An internal file is an area of storage internal to the program. An internal file can be a character variable, a character array element, a character array, or a character substring. Each variable, substring, or array element is one record; if the file is an array, each array element is one record. For example, the statement:

```
CHARACTER*15 city(50)
```

defines a character array containing 50 elements of 15 characters each. The array can be referenced as an internal file named `city` containing 50 records of 15 characters each.

You cannot use the `OPEN` or `CLOSE` statement, any file positioning statements, or any file status statements with internal files.

Reading From an Internal File

Data is usually read from internal files with a formatted `READ` statement. The name of the internal file is specified on the `READ` statement. For example, the `READ` statement in the following program reads four records from the internal file `course` and stores the data into variables `a`, `b`, `c`, and `d`.

```
PROGRAM internal_read1

CHARACTER*10 course(4), a, b, c, d
DATA course / 'chemistry', 'biology', 'zoology', 'botany' /

C   Read from internal file 'course'
   READ(course, '(A10)') a, b, c, d

   WRITE(6,*) a
   WRITE(6,*) b
   WRITE(6,*) c
   WRITE(6,*) d

STOP
END
```

The output of this program is:

```
chemistry
biology
zoology
botany
```

You can use the READ statement to convert data. For example, consider the following program:

```
PROGRAM internal_read2

CHARACTER int_file*20, string*20
INTEGER   a, b, c, d, e
DATA      string /' 31 61 4 18 91'/

C Assign the character string to the internal file:
  int_file = string

C Read the internal file and convert the data type:
  READ(int_file, '(5I4)') a, b, c, d, e

C Check the contents of the file:
  WRITE(6,*) 'a = ', a
  WRITE(6,*) 'b = ', b
  WRITE(6,*) 'c = ', c
  WRITE(6,*) 'd = ', d
  WRITE(6,*) 'e = ', e

END
```

This program performs the following: reads the character data from the internal file `int_file`, converts the character string into internal integer format, and stores the converted data into the variables `a`, `b`, `c`, `d`, and `e`. The output of the program is shown below:

```
a = 31
b = 61
c = 4
d = 18
e = 91
```

As an HP extension to the FORTRAN 77 standard, list-directed READ operations can use internal files.

Writing to an Internal File

Data is written to internal files with a formatted WRITE statement. The name of the internal file is specified in the WRITE statement. For example, the statement:

```
WRITE(UNIT=address_var, FMT='(I10)') street_address
```

or:

```
WRITE(address_var, '(I10)') street_address
```

writes the value of `street_address` into the first ten positions of the internal file `address_var`. The variable `address_var` must be a variable or array of type CHARACTER. If `address_var` has a length greater than ten, the rest of the record is filled with blanks.

The WRITE statement in the program below defines an internal file name and writes five records to the file.

```
PROGRAM internal_write

CHARACTER*14 name(5)
INTEGER      a, b, c, d, e
DATA        a/14/, b/57/, c/0/, d/-123/, e/95/
```

C Write to the internal file:

```
WRITE(name, '(1X, I4)') a, b, c, d, e
```

```
END
```

After the program executes, the array `name` is assigned the following values (the values are 14 characters long):

Element	Value
<code>name(1)</code>	14
<code>name(2)</code>	57
<code>name(3)</code>	0
<code>name(4)</code>	-123
<code>name(5)</code>	95
	<-----14 characters long----->

Another example of writing to a character variable in an internal file is shown below, where a format specification is built at execution time. This program only shows how internal files work, not necessarily efficient programming practices.

```
PROGRAM internal_write2

CHARACTER*14  ifmt
INTEGER      iarray(5)
DATA         iarray/1, 2, 3, 4, 5/

C Prompt for format specification variables:
  WRITE(6,*) 'Enter number of spaces before the array contents:'
  READ(5,*) n
  WRITE(6,*) 'Enter the repetition factor: '
  READ(5,*) m

C Create the internal file containing the format:
  WRITE(ifmt,10) n, m
10  FORMAT (1X, '(', I2, 'X,' I2, '(I2,X)')

C Print the array using the format in the internal file:
  WRITE(6,*) 'The contents of the array "iarray" is: '
  WRITE(6,ifmt) iarray

C Print the format used to store the data:
  WRITE(6,*) 'The format used to print the contents is: '
  WRITE(6,*) 'FORMAT ', ifmt

END
```

The program prompts you for portions of the FORMAT statement; the WRITE statement then writes the variables `n` and `m` to the character string in the internal file. Therefore, when the WRITE statement executes, the format specification will have the desired format.

A sample run of the program looks like this:

```
Enter number of spaces before the array contents: 10
Enter the repetition factor: 5
The contents of the array "iarray" is: 1 2 3 4 5
The format used to print the contents is: FORMAT (10X, 5(I2,X))
```

As an HP extension to the FORTRAN 77 standard, list-directed WRITE operations can use internal files.

HP FORTRAN 77/iX File Operations

This chapter describes the following:

- Characteristics requested by the OPEN statement processor
- Predefined units and files
- Creating and closing files
- Carriage control files
- Magnetic tapes
- Procedures for file handling

The OPEN Statement Processor

The FORTRAN 77 OPEN statement gives you more control over file connection and file characteristics than older versions of FORTRAN. The run-time library translates the OPEN statement to call the MPE/iX FOPEN system intrinsic. The OPEN statement processor implements the options specified in the OPEN statement by assigning the options to corresponding options in the FOPEN intrinsic. The options in the OPEN statement do not override the characteristics of an existing file; that is, a FILE equation pertaining to the file takes precedence over the FOPEN arguments. Any FOPEN option not specified by FORTRAN 77 is supplied with the file system defaults.

The options specified in the OPEN statement determine the values of the FOPTIONS and AOPTIONS arguments to the FOPEN intrinsic. The default FILESIZE (number of records) is 4096 and the default NUMEXTENTS (number of extents allowed) is 32. Because both values are four times greater than the MPE/iX default, you do not need a FILE equation with a file of more than 1023 records. If the MPE/iX defaults are used, no additional disc space is allocated for small files.

The name or formal file designator for a file is created by joining the characters FTN with the two-digit FORTRAN logical unit (LU) number. For example, file 8 is FTN08 and file 10 is FTN10.

Predefined Units and Files

Units five and six are the predefined I/O units for FORTRAN 77. These units are opened by the I/O library before the first I/O statement is executed. Unit five is opened as formal file designator FTN05, which defaults to \$STDINX. (\$STDIN is not the default; this prevents logging off if an input line of :EOF is entered and allows input of lines containing a colon in the first position.) The formal designator for unit 6 is FTN06, which defaults to \$STDLIST. Both FTN05 and FTN06 are first opened by FOPEN as MPE/iX OLD or TEMP files so they can be redirected by a :FILE equation. If no file or file equation exists for FTN05 or FTN06, another FOPEN is executed to open the files as NEW MPE/iX files.

For example, using the equation

```
:FILE FTN06 = outfile
```

with the FORTRAN 77 statement

```
WRITE(6,*) " This will be diverted into the file 'outfile'."
```

causes the output to be written to the file `outfile` instead of to the terminal.

FTN05 Unit five is first opened with FOPTIONS of octal 157, which opens \$STDINX as an OLD or TEMP ASCII file with variable length records. If there is no existing file (or file equation) for FTN05, the first FOPEN fails and another FOPEN is attempted with FOPTIONS of octal 154. Octal 154 has the same file characteristics as octal 157, except a NEW file is requested. The AOPTIONS for both calls are octal 300, requesting read-only, shared access to the file. Shared access is requested in case FORTRAN 77 I/O is performed from subprograms called from another language that might have already opened \$STDINX. Access is denied if exclusive access was requested.

FTN06 Unit six is first opened with FOPTIONS of octal 517, which opens \$STDLIST as an OLD or TEMP ASCII file with variable length records and with carriage control. If there is no existing file (or file equation) for FTN06, the first FOPEN call fails and another FOPEN is attempted with FOPTIONS of octal 514. Octal 514 has the same file characteristics as octal 517, except a NEW file is requested. The AOPTIONS for both calls are octal 301, requesting write-only, shared access to the file.

**FTN01 Through FTN99
(Excluding FTN05 and
FTN06)**

For compatibility with FORTRAN 66/V, the FORTRAN 77 I/O library automatically opens units 1 through 99 (excluding 5 and 6) to the formal file designators FTN01 through FTN99, respectively. OPEN statements are not required for these files, though a :FILE equation is usually required (as it is in FORTRAN 66/V).

Units 0 and those greater than 99 must be opened with an OPEN statement before being used.

Creating Files with the OPEN Statement

Existing files (OLD or TEMP) connected with the OPEN statement already have defined characteristics; therefore, the arguments to the FOPEN intrinsic are ignored. Similarly, FILE equations referenced in the OPEN statement are overridden by the characteristics of the corresponding device or file.

By default, files connected by the OPEN statement processor besides FTN05 and FTN06 are opened by FOPEN with AOPTIONS of 4, requesting read/write access. This prepares the system for any arbitrary mix of READ and WRITE statements. As an extension to the ANSI standard, the READONLY specifier on OPEN is allowed. Such files are opened by FOPEN with AOPTIONS of 0, requesting readonly access. OPEN statements corresponding to files (or file equations) that can only be read or written are valid unless an operation is requested that is not allowed on the file. Invalid requests cause a file system error.

STATUS='NEW'

When the OPEN statement specifies **STATUS='NEW'**, the FORTRAN compiler makes sure that the referenced file does not exist by attempting to open the file as an MPE/iX OLD or TEMP file. If the file exists, the file is closed and an error is reported. Normally, the first FOPEN fails because no file exists and the FOPEN is tried again with FOPTIONS requesting an MPE/iX NEW file. This attempt usually succeeds in creating the file; if not, a file system error is reported.

STATUS='OLD'

When the OPEN statement specifies **STATUS='OLD'**, the corresponding FOPEN intrinsic specifies to only search the MPE/iX OLD or TEMP file domains. The temporary job domain is searched first; therefore, TEMP files are connected before permanent (OLD) files of the same name. If the FOPEN fails (no such named file exists), an error is reported.

STATUS='SCRATCH'

When the OPEN statement specifies **STATUS='SCRATCH'**, the file is opened as a nameless MPE/iX file. Therefore, in accordance with the ANSI standard, it is impossible to save the file. If the file is not a scratch file, the file is defined by the FORM and ACCESS options in the OPEN statement.

STATUS='UNKNOWN'

OPEN statements that omit the STATUS specifier default to **STATUS='UNKNOWN'**. **STATUS='UNKNOWN'** is implemented as if OLD were specified, with the exception that if the first FOPEN fails, the open is reattempted with NEW status.

FORM='UNFORMATTED'
and
FORM='FORMATTED'

The FORM option specifies the type of transfers that can be performed on the file. FORM='UNFORMATTED' implies that only unformatted (binary) transfers are performed, so an MPE/iX BINARY file is requested in the corresponding FOPEN intrinsic. When the OPEN statement specifies FORM='UNFORMATTED', bit 13 is cleared to zero in the FOPTIONS parameter.

The default FORM='FORMATTED' implies that only formatted and/or list-directed transfers can be performed. Accordingly, an MPE/iX ASCII file is requested by setting bit 13 in the FOPTIONS parameter. Attempting a transfer type not allowed on the associated file results in an error.

AC-
CESS='SEQUENTIAL'

Sequential READ and WRITE statements can have I/O lists of records with varying lengths; in fact, the ANSI standard does not specify an upper limit to the length of a sequential record. To efficiently implement varying lengths, use MPE/iX variable record length files as the default sequential file type (FOPTIONS bit eight cleared to zero and bit nine set). This implies that files opened for sequential access cannot be accessed directly because it is impossible to access variable record length files by FREADDIR or FWRITEDIR. The record length requested for sequential files is zero, implying that the MPE/iX default of the configured physical record size of the device is to be used (256 bytes on disk files). Also, on variable record length files created by the OPEN statement, the MPE/iX default becomes the maximum logical record length. The default can be overridden by using a FILE equation that specifies a longer record length. For example, the command

```
:FILE outfile; REC = -5120,,V,ASCII
```

specifies that variable length records up to 5120 bytes long can be written or read.

The FCONTROL intrinsic that implements the BACKSPACE statement for files of fixed and undefined record lengths does not apply to variable record length files. Therefore, variable record length files must be backspaced by rewinding and then reading forward to the previous record. Even though the FORTRAN 77 library performs the rewinding and reading forward, this is clearly not a performance feature of the implementation. A file equation to specify a new file as fixed record length type will save execution time for programs that often backspace.

ACCESS='DIRECT'

OPEN statements that specify **ACCESS='DIRECT'** must include the maximum length of the records to be read or written. This allows fixed record length files of the appropriate length to be requested in the FOPEN. Accordingly, bits eight and nine of the FOPTIONS are set to zero for the FOPEN call. The required RECL specifier sets the record length of the file created. If an odd number of bytes is requested for the RECL option on a direct unformatted file, MPE/iX rounds the length up to the next higher word length (the byte count becomes even). This is done because binary files are strictly defined in terms of 16-bit words. The increased byte count is returned if the INQUIRE statement requests the record length.

Closing Files

Once opened by predefinition or with the OPEN statement, files are closed either by executing a CLOSE statement or by terminating the program.

Executing the CLOSE Statement

The CLOSE statement explicitly causes the corresponding unit to be closed. The disposition of the file is controlled by the STATUS specifier in the CLOSE statement, with the exception of STATUS='SCRATCH' files. If the status is not specified in the CLOSE statement, named files default to KEEP (kept as an MPE/iX permanent file) and scratch files default to DELETE. An error occurs if you attempt to save a scratch file.

Note



When a sequential file is closed, the last record written to the file is the last record of the file.

Files opened in the temporary job domain are closed as permanent files. If the FCLOSE fails (such as if the permanent file name already exists), the FCLOSE is attempted again in the temporary domain. An error occurs if the second FCLOSE fails.

Terminating a Program

All files remaining open when your program terminates are closed either by the FORTRAN 77 library or by the MPE/iX operating system. The disposition of files is determined by the type of program termination. There are two types of termination: normal and error.

Normal termination occurs when a program executes a STOP or END statement in the main program. Both statements close all opened files as if a CLOSE statement requesting default disposition had been executed on the corresponding units.

Error termination occurs when a program terminates because of errors detected by the FORTRAN 77 library, the Compiler Library, or by the MPE/iX operating system. Errors found by the FORTRAN 77 library (primarily I/O errors) that are not trapped by the ERR or IOSTAT specifiers in the READ and WRITE statements cause error messages to be printed. Errors detected by the Compiler Library (primarily math function errors) or by MPE/iX cause error messages to be printed. Any open files are closed by the MPE/iX operating system, not by the FORTRAN 77 library. When closed by MPE/iX, an FCLOSE with MPE/iX default status (not FORTRAN 77 status) is performed, which closes files in the domain in which they were opened. Because files are always created in the NEW file domain, such an FCLOSE operation causes newly created files to be deleted.

Carriage Control Files

The preconnected unit six (FTN06) is opened with carriage control (CCTL). When initially opening empty files (EOF points to zero), the CCTL bit and the device type are checked. The appropriate FWRITE is then performed to set the file for prespacing. Prespacing mode is when the carriage control character is in column one of the formatted output.

Terminals and Line Printers

When the file is a terminal or line printer, the device is set into prespacing mode by performing an FWRITE of length zero with carriage control option of 101 octal. Because the carriage control is immediately executed on these devices, the control code does not have to be actually written as data to the file.

Disk Files

If a carriage control disk file is opened and found to be empty, the I/O library writes the carriage control option of 101 octal (ASCII "A") into the file. This allows the file to be later copied to a CCTL device and still retain the carriage control. The I/O library performs an FWRITE of one byte length to embed the prespacing code. For example, the following statement is valid:

```
WRITE(12, '("A")')
```

FILE Equation

If you want to create a CCTL file on a unit other than FTN06, you must provide a file equation for the opened file that includes the CCTL characteristic. A CCTL FILE command for a file that already exists without carriage control in the file label is overridden by the existing file's characteristics. For example, the equation

```
:FILE outfile; CCTL
```

with the FORTRAN 77 statements

```
OPEN(16, FILE='outfile')
```

```
WRITE(16,100) " This will be written at the top of a page."  
100 FORMAT ('1', A)
```

causes the output to be written to the disk file `outfile` with carriage control preserved. When the file is copied to a carriage control output device, the output line is printed at the top of a page.

Using Magnetic Tapes

Magnetic tapes can be directly read or written from FORTRAN 77 programs by a FILE equation. As a MIL-SPEC 1753 extension to the ANSI standard, such tapes can be read or written in multiple logical files on the same tape. The most portable format is a fixed record length ASCII file with a specified blocking factor of one. However, the blocking factor of one wastes tape because the interrecord gaps are longer on the tape than on a single formatted record of normal length. For example, the equation

```
:FILE OUTFILE; DEV=TAPE; REC=-80,10,F,ASCII
```

with the FORTRAN 77 statements

```
OPEN(16,FILE='OUTFILE')
```

```
DO i = 1,400
```

```
  WRITE(16,*) " This will be written on a tape 400 times."
```

```
ENDDO
```

causes a tape file to be created with ASCII logical records 80 bytes long in blocks of 10 logical records per physical record.

FORTRAN/3000 programs that use the UNITCONTROL intrinsic are supported by HP FORTRAN 77/iX. All the options of the FORTRAN/3000 version are supported on HP FORTRAN 77/iX, including those operations commonly used for magnetic tape handling.

Using the File Handling Procedures

This section describes the FSET, FNUM, and UNITCONTROL procedures. Refer to the *HP FORTRAN 77/iX Reference Manual* for more details on these procedures.

FSET Procedure

The FSET procedure changes the MPE/iX operating system file number assigned to a given FORTRAN logical unit number in the file table.

The FSET procedure can be called from a FORTRAN 77 program as follows:

```
CALL FSET(unit,newfile,oldfile)
```

Item	Description/Default	Restrictions
unit	Positive integer constant or variable (INTEGER*2 or INTEGER*4) that specifies the FORTRAN file unit for which the change is to be made.	Must be nonnegative.
newfile	Positive integer constant or variable (INTEGER*2 or INTEGER*4) that specifies the new MPE/iX file number to be assigned to unit .	Must be nonnegative.
oldfile	Integer variable to which the procedure returns the old value of the file number that was assigned to unit .	Cannot be the same variable as newfile .

The following program shows how to use the FSET procedure to assign a FORTRAN logical unit number.

```
$WARNINGS OFF
$SHORT

      PROGRAM fset_example
C
C  FOPEN, FSET, and FCLOSE Example

      IMPLICIT NONE
      INTEGER filenumber,oldnum
      SYSTEM INTRINSIC FOPEN,FCLOSE
      CHARACTER buffer*72,filename*16
      PARAMETER (FILENAME = 'maillist')

      filenumber = FOPEN(filename,1B,105B)
      IF (ccode())30,10,30

C  Call FSET to assign the FORTRAN unit number five to "filenumber"

10    CALL FSET(5,filenumber,oldnum)
      PRINT *,'Old file number = ',oldnum
      PRINT *,'FOPEN number = ',filenumber
20    READ(5, '(A72)',END=40)buffer      ! Read to EOF
      WRITE(6,100)buffer(1:19)
100   FORMAT(T2,A20)
      GO TO 20
30    PRINT *,'Could not open file'
      STOP

C  Close the file

40    CALL FCLOSE(filenumber,1B,0B)
      IF (ccode())50,60,50
50    PRINT *,'Could not close file'
      STOP
60    PRINT *,'File closed successfully'
      STOP
      END
```

This is the output of the program:

```
Old file number = 11
FOPEN number = 10
SMILEY    FACE
MICKEY    MOUSE
SLIM      JIM
CHARITY   BELL
DONALD    DUCK
JOE       SMOE
CLAIRE    PLIMSOL
INDIANA   JONES
JAKE      FAKE
File closed successfully
```

FNUM Procedure

The FNUM procedure returns the MPE/iX system file number assigned to a given FORTRAN 77 logical unit number. This procedure returns an INTEGER*2 or INTEGER*4 value.

The FNUM procedure can be called from an HP FORTRAN 77/iX program as an external function as follows:

$$I = \text{FNUM}(unit)$$

Item	Description/Default	Restrictions
unit	Positive integer (INTEGER*2 or INTEGER*4) that specifies the FORTRAN file unit for which the MPE/iX system file number is to be extracted. The value returned is the same size as the argument.	Must be nonnegative.

See the UNITCONTROL procedure description below for an example of using the FNUM procedure.

UNITCONTROL Procedure

The UNITCONTROL procedure allows an HP FORTRAN 77/iX program to request several actions (see below) for any FORTRAN 77 logical unit.

The UNITCONTROL procedure is called as follows:

```
CALL UNITCONTROL(unit,opt)
```

Item	Description/Default	Restrictions
unit	Positive integer (INTEGER*2 or INTEGER*4) that specifies the unit number of the file to be used.	Must be nonnegative.
opt	Integer (INTEGER*2 or INTEGER*4) that specifies the option (see Table 4-1).	None.

The options for the UNITCONTROL intrinsic are listed in Table 4-1.

Table 4-1. UNITCONTROL Options

Option	Description
-1	Rewind (but don't close the file)
0	Backspace
1	Write an EOF mark.
2	Skip backward to a tape mark
3	Skip forward to a tape mark
4	Unload the tape and close the file
5	Leave the tape loaded and close the file
6	Convert the file to prespacing
7	Convert the file to postspacing
8	Close the file

Note



Use the REWIND, BACKSPACE, ENDFILE, and CLOSE statements instead of the -1, 0, 1, and 8 options. These statements are part of the FORTRAN 77 language and are more portable.

Option values outside the range of -1 through 8 are ignored and no action is taken.

The program below shows how to use the FNUM and UNITCONTROL procedures. The SHORT compiler directive forces the integer and logical default size to two bytes.

```
$WARNINGS OFF
$SHORT
PROGRAM unit_fnum_ex

C Example program using FNUM and UNITCONTROL

IMPLICIT NONE
SYSTEM INTRINSIC HPMERGEINIT
CHARACTER buffer*72
INTEGER*4 keysonly,numkeys,keys(8),infiles(3),status
INTEGER*4 outputfile(2),error

C Merge two sorted files, MAIL1 (unit 20) and MAIL2 (unit 21)
C into a third file, MAIL3 (unit 22)

C Open all files

OPEN(20,FILE='mail1',STATUS='OLD',ERR=200)
OPEN(21,FILE='mail2',STATUS='OLD',ERR=300)
OPEN(22,FILE='mail3',STATUS='NEW',ERR=400)

C Establish keys for SORT - major at column 11 for 9 bytes
C (LAST NAME) and minor at column 1 for 10 bytes (FIRST NAME)

keys(1) = 11
keys(2) = 9
keys(3) = 10
keys(4) = 0
keys(5) = 1
keys(6) = 10
keys(7) = 10
keys(8) = 0

keysonly = 0
numkeys = 2

C Establish MPE/iX filenumbers for input files (MAIL1 and MAIL2)
C by referencing the FNUM procedure

infiles(1) = FNUM(20)
infiles(2) = FNUM(21)
infiles(3) = 0
```

```

C Establish MPE/iX filenumbers for output file (MAIL3) by
C referencing the FNUM procedure

      outputfile(1) = FNUM(22)
      outputfile(2) = 0

C Merge the files

      CALL HPMERGEINIT(status,infiles,,outputfile,,
>                    keyonly,numkeys,keys,,,,error)
      IF (error .NE. 0) STOP 'Merge failed'

C Display the new merged file

      REWIND 22
20    READ(22,'(A72)',END=30) buffer
      WRITE(6,100)buffer
100   FORMAT(T2,A)
      GO TO 20

C Call UNITCONTROL to close the files MAIL1, MAIL2, and MAIL3,
C which is the same as using the CLOSE statement

30    CALL UNITCONTROL(20,8)
      CALL UNITCONTROL(21,8)
      CALL UNITCONTROL(22,8)
      STOP

200   PRINT *,'Could not open file - MAIL1'
      STOP
300   PRINT *,'Could not open file - MAIL2'
      STOP
400   PRINT *,'Could not open output file - MAIL3'
      END

```

This is the output of the program:

PLAINS	ANTELOPE	201	OPENS	SPACE	AVE	BIGCOUNTRY	WY
LOIS	ANYONE	6190	COURT	ST		METROPOLIS	NY
KING	ARTHUR	329	EXCALIBUR	ST		CAMELOT	CA
ALI	BABA	40	THIEVES	WAY		SESAME	CO
BLACK	BEAR	47	ALLOVER	DR		ANYWHERE	US
JOHN	BIGTOWN	965	APPIAN	WAY		METROPOLIS	NY
KNEE	BUCKLER	974	FISTICUFF	DR		PUGILIST	ND
SWASH	BUCKLER	497	PLAYACTING	CT		MOVIETOWN	CA
ANIMAL	CRACKERS	1000	CRUNCH	LN		COOKIE	US
MULE	DEER	963	FOREST	PL		HIGHCOUNTRY	CA
WHITETAIL	DEER	34	WOODSY	PL		BACKCOUNTRY	ME
JAMES	DOE	4193	ANY	ST		ANYTOWN	MD
JANE	DOE	3959	TREEWOOD	LN		BIGTOWN	MA
PRAIRIE	DOG	493	ROLLINGHILLS	DR		OPENS	ND
JOHN	DOUGHE	239	MAIN	ST		HOMETOWN	MA
MALLARD	DUCK	79	MARSH	PL		PUDDLEDUCK	CA
JENNA	GRANDTR	493	TWENTIETH	ST		PROGRESSIVE	CA
KARISSA	GRANDTR	7917	BROADMOOR	WAY		BIGTOWN	MA
SNOWSHOE	HARE	742	FRIGID	WAY		COLDSPOT	MN
MOUNTAIN	LION	796	KING	DR		THICKET	NM
SPACE	MANN	9999	GALAXY	WAY		UNIVERSE	CA
SWAMP	RABBIT	4444	DAMPLACE	RD		BAYOU	LO
NASTY	RATTLER	243	DANGER	AVE		DESERTVILLE	CA
BIGHORN	SHEEP	999	MOUNTAIN	DR		HIGHPLACE	CO
GREY	SQUIRREL	432	PLEASANT	DR		FALLCOLORS	MA

Subprograms

A subprogram is an independent section of code called by a main program or by another subprogram. Subprograms make programs more readable and easier to maintain, write, and debug.

Subprograms can be grouped into these three categories:

- Subroutines
- Functions
 - Function subprograms
 - Statement functions
 - Intrinsic functions
- Block Data Subprograms

A program unit, such as a main program or a subprogram, is a sequence of FORTRAN statements. Table 5-1 summarizes the components of program units.

Table 5-1. Components of Program Units

Component	Description	How Identified
Main program	Defines the main entry point.	Can begin with the PROGRAM statement.
Subroutine	Returns values through argument lists or common blocks.	Begins with the SUBROUTINE statement.
Function	Returns values through the function name, argument lists, or common blocks.	Begins with the FUNCTION statement.
Statement function	Calculates a single result; cannot be referenced outside of the program unit that defines it.	Defined in a program unit.
Block data subprogram	Provides initial values for named and blank common blocks.	Begins with the BLOCK DATA statement.

This chapter describes subroutine, function, and block data subprograms.

Subroutines

Subroutines are user-written procedures that perform a computational process or a subtask for another program unit. Subroutines usually perform part of an overall task, such as solving a mathematical problem, performing a sort, or printing in a special format. Values are passed to the subroutine and returned to the calling program unit by using arguments or common blocks.

Structure of a Subroutine

The first statement of a subroutine must be a SUBROUTINE statement. Here are some examples of SUBROUTINE statements:

```
SUBROUTINE next(arg1, arg2)
```

```
SUBROUTINE last(a, *, *, b, i, k, *)
```

```
SUBROUTINE noarg
```

The subroutine names are `next`, `last`, and `noarg`. Values are passed to a subroutine by dummy arguments (`arg1`, `arg2`, `a`, `b`, `i`, and `k` in the above examples) or common blocks. Dummy arguments are also called formal arguments. Dummy arguments, common blocks, and asterisks are explained later in this chapter.

A subroutine can contain any statement except another SUBROUTINE statement, or a BLOCK DATA, FUNCTION, or PROGRAM statement. As an extension to the ANSI 77 standard, HP FORTRAN 77 subroutines can be recursive. That is, a subroutine can call itself either directly or indirectly. For example, in the program below, the subroutine `rsub1` directly calls itself.

```
PROGRAM recursive      ! Main program
INTEGER count

count = 0

CALL rsub1(count)
PRINT *, 'final count = ', count

END

SUBROUTINE rsub1(num)  ! Subroutine rsub1

IF (num .LT. 5) THEN
    num = num + 1
    PRINT *, 'num = ', num
    CALL rsub1(num)    ! rsub1 directly calls itself
END IF

END
```

The program produces the following output:

```
num = 1
num = 2
num = 3
num = 4
num = 5
final count = 5
```

A program that indirectly calls itself is similar in principle to a subroutine that calls a procedure that in turn calls the original subroutine.

The END statement in a subroutine causes control to be passed back to the calling program.

The RETURN statement also transfers control back to the calling program. You only need to use RETURN statements for returning to the calling program from a place other than the end of a subprogram. When the RETURN statement in the subroutine is executed, control normally returns to the statement following the CALL statement in the calling program. If necessary, there can be several RETURN statements in a subprogram.

The STOP statement in a subroutine terminates program execution. For example, the output of the program:

```
PROGRAM stopit
CALL sub
PRINT *, 'Hello'
END

SUBROUTINE sub
PRINT *, 'Goodbye'
STOP
END
```

is:

```
Goodbye
```

Invoking Subroutines

A subroutine is executed when a `CALL` statement is specified in a program unit. Here are some examples of `CALL` statements:

```
CALL next(x, y)
```

```
CALL last(a, *10, *20, b, i, k, *30)
```

```
CALL noarg
```

When the subroutine is executed, the actual arguments `x`, `y`, `a`, `b`, `i`, and `k` in the `CALL` statement are associated with their equivalent dummy arguments in this way:

```
PROGRAM main
.
.
.
CALL    sub1(actual_arg1, actual_arg2, actual_arg3)
END
                ↑           ↑           ↑
SUBROUTINE sub1(dummy_arg1, dummy_arg2, dummy_arg3)

END
```

The subroutine is then executed using the actual argument values. Arguments can be variable names, array names, array elements, record names, record field names, constants, and expressions.

Values can also be passed to a subroutine by specifying an alternate return form using asterisks. The use of arguments and asterisks is described later in this chapter.

Alternate Returns From Subroutines

Normally control from a subroutine returns to the calling program unit at the statement following the CALL statement. However, you can specify an alternate return that allows control to return to the calling program unit at any labeled executable statement.

An alternate return is specified in the called subroutine by the RETURN statement with an integer expression or constant that identifies the number of a statement label in the CALL statement. The SUBROUTINE statement must contain one or more asterisks (*) or ampersands (&) corresponding to alternate return labels in the CALL statement. An example of a CALL statement and its associated SUBROUTINE and alternate return statements is shown below.

```
PROGRAM alternate
  n = 2
  CALL sub(n, *10, *20, *30)
10  i = 1
    GO TO 50
20  i = 2
    GO TO 50
30  i = 3
50  PRINT *, 'i = ', i
    PRINT *, 'n = ', n
    END

SUBROUTINE sub(n, *, *, *)
  RETURN n      ! Return to the nth statement
  END
```

Control returns to the main program from the subroutine as follows:

Returns To Statement	With This Value of <i>n</i>
10	1
20	2
30	3

In this example, because *n* is equal to two, control returns to statement 20. The value of *i* will be set to two. The output from the program looks like this:

```
i = 2
n = 2
```

If the RETURN statement contains an expression, the value of the expression cannot exceed the number of asterisks in the SUBROUTINE statement. Also, for ease of understanding and portability, the number of asterisks in the SUBROUTINE should equal the number of alternate return labels specified in the CALL

statement. If an expression in a RETURN statement has a value that is either less than one or greater than the number of alternate return labels in the CALL statement, control is returned to the statement following the CALL statement.

An example of a program that uses alternate returns follows. The subroutine searches a file named `parts` to validate a part number. Each record in `parts` is an integer array of two elements; the first is the part number and the second is a code. A negative code indicates an obsolete part number. All existing part numbers are in the file `parts`. The records in the file are ordered by increasing part number and, for simplicity, the search for a part number is sequential. The return from the subroutine to the main program is summarized below:

Condition	Type of Return
Part number is found and the part number is not obsolete	Normal return to main
Part number is obsolete	First alternate return is taken
Part number is not found	Second alternate return is taken

```

PROGRAM prog
  INTEGER part_number

C  Get a part number
  PRINT *, 'Enter part number '
  READ *, part_number

  CALL validate(part_number, *100, *200)

C  Normal return.  Process for valid part number.
  PRINT *, part_number
  GO TO 999

C  First alternate return.  Process for obsolete part number.
  100 PRINT *, 'Obsolete part number = ', part_number
  GO TO 999

C  Second alternate return.  Process for invalid part number.
  200 PRINT *, 'Invalid part number = ', part_number

999  END

SUBROUTINE validate(part_number, *, *)
  INTEGER parts_file_record(2), part_number
  LOGICAL obsolete_flag, part_found_flag

C  Initialize variables
  obsolete_flag = .FALSE.
  part_found_flag = .FALSE.
  parts_file_record(1) = 0

```

```

C Search for part number and set flags accordingly
  OPEN(111, FILE='parts', STATUS='OLD')

  DO WHILE (parts_file_record(1) .LT. part_number)
    READ(111,*) parts_file_record
    IF (parts_file_record(1) .EQ. part_number) THEN
      part_found_flag = .TRUE.
      IF (parts_file_record(2) .LT. 0) obsolete_flag = .TRUE.
    ENDIF
  END DO

  CLOSE(111)

C Return to calling program depending on flags
  IF (obsolete_flag) RETURN 1
  IF (.NOT. part_found_flag) RETURN 2
  RETURN
END

```

Functions

FORTRAN functions can be grouped into categories, as summarized in Table 5-2.

Table 5-2. Categories of FORTRAN Functions

Type of Function	Description
Function Subprogram	A user-defined function
Statement Function	A user-defined single statement function
Intrinsic Function	A FORTRAN built-in function

A function name in an expression causes the function to be evaluated; the function then assigns a value to the function name. As with a subroutine, a function can also return values through its arguments or through common blocks. However, these *side effects* should be avoided because they inhibit clarity. The effect of a function should be the calculation of a single result returned through the function name.

Function Subprograms

A function subprogram is a user-written FORTRAN function in a program. A function is invoked by using the function name, followed by the argument list.

The FUNCTION statement is the first statement of a function. Here are some examples of FUNCTION statements:

```
FUNCTION time()
```

```
INTEGER*4 FUNCTION add(k, j)
```

```
LOGICAL FUNCTION key_search(char_string, key)
```

The function names are: `time`, `add`, and `key_search`. Values are passed to function subprograms by arguments (`k`, `j`, `char_string`, and `key` in the statements above are arguments) or common blocks. An argument list is not required, but you must use parentheses to differentiate the function name from a simple variable.

An example of a user-defined function is shown below:

```
PROGRAM main

READ (5,'(2F10.0)') a, b

x = bigger(a, b)

WRITE (6,100) a, b, x
100 FORMAT (1X, 2F10.2, /, 1X, 'The largest value is', F10.2)
END

FUNCTION bigger(a, b)      ! Function to return the larger value
IF (a .LT. b) THEN
    bigger = b
ELSE
    bigger = a
ENDIF

END
```

A function can contain declaration, assignment, input/output, and flow control statements; a function cannot contain another FUNCTION statement, a BLOCK DATA, a SUBROUTINE, or a PROGRAM statement. As an extension to the ANSI 77 standard, an HP FORTRAN 77 function subprogram can be recursive. That is, a function can contain a direct or indirect reference to itself.

The END statement transfers control back to the calling program where the function call was made. The function subprogram always returns to the expression from which it was invoked. Alternate returns are not allowed in function subprograms.

The RETURN statement also transfers control back to the calling program. You only need to use RETURN statements for returning to the calling program from a place other than the end of a function. The last line of a function must be an END statement.

To associate a value with the function subprogram name, the function name must be used within the function in one or more of these ways:

- Specified on the left side of an assignment statement
- Included as an element of an input list in a READ statement
- Be an actual argument of a function or subroutine reference

Some examples demonstrating how to associate a value to the function name are shown below. Consider this function:

```
        INTEGER FUNCTION factorial(n)
        INTEGER fact, n
        fact = 1

        DO 10 i = 2,n
            fact = fact * I
10     CONTINUE

        factorial = fact

    END
```

In the function `factorial` above, the value `fact` is assigned to the function name `factorial`. Note that the DO loop will not be executed if `n` equals zero or one.

Here is another function:

```
        FUNCTION tot(num,sum)
        REAL num, sum

        IF (num .GE. 0.0) THEN
            tot = sum + num
        ELSE
            READ (5,*) tot
        ENDIF

    END
```

In the function `tot` above, a value is assigned to the function name `tot` in one of two ways: by appearing on the left side of an assignment statement or by appearing in the input list of a `READ` statement.

Finally, look at the function `next1`:

```
        FUNCTION next1(back)

        IF (back .GT. 1.5) THEN
            CALL gtfwrđ(next1)
        ELSE
            CALL gtback(next1)
        ENDIF

    END
```

The function `next1` shows how a function name is associated with a value in one of two subroutines. Within the subroutines, `next1` must be assigned a value.

Because a value is assigned to the function subprogram name, the value's data type must be defined. The data type associated with the function name is determined in one of these ways:

- If the data type is included with the FUNCTION statement, the name is assigned that type. For example, a FUNCTION statement explicitly specified as an integer looks like this:

```
INTEGER FUNCTION funcname()
```

A function name cannot have the data type specified more than once in a program. For example, using the following statements together is illegal:

```
INTEGER FUNCTION funcname()
```

```
INTEGER funcname
```

- If the data type is not included in the FUNCTION statement, the function name can be declared in a type statement within the function. The type statements are: CHARACTER, COMPLEX*8, COMPLEX*16, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8, and REAL*16, as well as BYTE, COMPLEX, DOUBLE COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, and REAL.

For example, the following statements define an integer function:

```
FUNCTION funcname()
```

```
INTEGER funcname
```

- If the data type is not included in the FUNCTION statement and is not declared in a type statement, the type is assigned implicitly according to the first letter of the function name. Unless modified by an IMPLICIT statement, function names beginning with the letters A through H and O through Z define a REAL data type; letters I through N define an INTEGER data type.

The data type of the value associated with the function name in each program unit must agree with the type of the function.

When you reference a character function, the length of the function must be the same as that declared in the function. There is always agreement of length if a length of asterisk (*) is specified in the function. For example, a character function and a program that calls the function are shown below. The function returns the character string with all preceding and trailing blanks removed.

```

PROGRAM test

CHARACTER*20 input_string, result, stringtrim

DO WHILE (input_string(1:1) .NE. '0')
  WRITE (6,'(A,NN)') 'Enter a string: '
  READ (5,'(A)') input_string
  result = stringtrim(input_string,length)
  WRITE (6,'(A1,A,A1)') ':', result(1:length), ':'
END DO
END

CHARACTER*(*) FUNCTION stringtrim(string,length)
CHARACTER*(*) string
INTEGER length, left, right, i, j
DO k= 1, LEN(stringtrim)
  stringtrim(k:k)=" " ! Initialize stringtrim
END DO

left = 1

right = LEN(string) ! The intrinsic function LEN returns
C the length of the string.

DO WHILE ((string(left:left) .EQ. ' ') .AND. (left .LT. right))
  left = left + 1
END DO

DO WHILE ((string(right:right) .EQ. ' ') .AND. (right .GT. left))
  right = right - 1
END DO

length = right - left + 1
DO 10 i = 1, length
  stringtrim(i:i) = string(left:left)
  left = left + 1
10 CONTINUE

C The default is to return one blank if a string is all blanks

END

```

A sample run, where Δ represents a blank, looks like this:

```
Enter a string: string
:string:
Enter a string:  $\Delta\Delta\Delta$ four
:four:
Enter a string: three $\Delta\Delta$ 
:three:
Enter a string:  $\Delta\Delta$ blanks $\Delta\Delta\Delta$   $\Delta$ 
:blanks:
Enter a string:  $\Delta\Delta\Delta\Delta\Delta$ 
: :
Enter a string: 0
:0:
```

The value returned by the function is the value last assigned to the function name at the time a RETURN statement is executed in the function.

Consider this example of a calling program unit and a function subprogram. The program asks for input of two numbers m and n and computes the combinations of m items taken n at a time. That is, the function computes the following:

$$\frac{m!}{n!(m-n)!}$$

The function subprogram `factorial` is invoked in the expression that calculates `result`.

```
PROGRAM main
INTEGER*4 factorial, result

WRITE (6,*) 'Enter m and n: '
READ (5,*) m, n
result = factorial(m) / (factorial(n) * factorial(m-n))
WRITE (6,'(1X, I5, " things taken ", I5,
c          " at a time = ", I8)') m, n, result
END

INTEGER FUNCTION factorial(num)
INTEGER fact, num

fact = 1
DO 10 i=2, num
    fact = fact * i
10 CONTINUE

factorial = fact

END
```

Two runs might look like this:

```
Enter m and n: 7, 4
    7 things taken    4 at a time =    35
```

```
Enter m and n: 10, 2
   10 things taken    2 at a time =    45
```

Statement Functions

A statement function is a user-defined single-statement computation that can only be called in the program unit that defines it. The form of a statement function is similar to an arithmetic, logical, or character assignment statement. Only one value is returned from a statement function.

A statement function is invoked just like a function subprogram. When your program calls a statement function, the dummy arguments are replaced by actual arguments within the function expression. For example, if you define the function `calculate` as:

```
calculate(x, y, z) = y * x * (y + x) - z
```

the statement:

```
result = a + calculate(a, b, c)
```

gives the same result as if you had written:

```
result = a + (b * a * (b + a) - c)
```

Following are some more examples of statements functions:

```
root(a, b, c) = (-b + SQRT(b * b - 4. * a * c)) / (2. * a)
```

```
disp(c, r, h) = c * 3.1416 * r * r * h
```

```
indexq(a,j) = IFIX(a) + j - ic
```

The statement function is called with its symbolic name and an actual argument list in an arithmetic, logical, or character expression. For example, the program below defines and calls a statement function.

```
PROGRAM functionex
```

```
root(a, b, c) = (-b + SQRT(b * b - 4. * a * c)) / (2. * a)
```

```
var1 = 2.0
```

```
var2 = -9.0
```

```
var3 = 4.0
```

```
var4 = root(var1, var2, var3)
```

```
END
```

The value of `var4` will be 4.0.

A statement function can call another statement function. For example, the program below defines a statement function that calls another statement function.

```
PROGRAM functionex2

  add(a, b, c) = a + b + c
  add25(d, e, f) = add(d, e, f) + 25.0
  DATA value1, value2, value3 /5.0, 10.0, 15.0/
  result = add25(value1, value2, value3)
  PRINT *, result

END
```

All statement function definitions must precede the first executable statement in the program unit and must follow any specification statements in a program unit. The name of a statement function cannot be the same as a variable name, an array name, or a record name in the same program unit.

All arguments in the dummy argument list are simple variables and assume the value of the actual arguments in the same program unit when the function is invoked; that is, dummy arguments are replaced by actual arguments. The actual arguments can be variables, constants, and expressions. Variables in the statement function not included in the argument list assume the current value of the variable name in the program unit. For example, in the statement function:

```
indexq(a, j) = IFIX(a) + j - ic
```

the variable `ic` is not an argument, but is an ordinary variable defined outside the statement function.

A call to a statement function does not cause control to “jump” to another section of code; instead, the compiler substitutes the statement function code into the program code. A statement function cannot call itself directly or indirectly; that is, statement functions are not recursive.

The data type of a statement function is determined in the same way as for a variable. The type is either declared explicitly in a type statement or determined implicitly by the function name. If the type of the statement function is not the same type as the expression to the right of the equal sign in the statement function and if the function name and expression are both numeric, both logical, or both character, the expression is converted to the type of the function. For example, the statement function:

```
f(i) = i + j
```

has integer variable `i` and `j` and a real function name `f`. The expression `i + j` is converted to real.

An intrinsic function is a built-in function that is available to your program. Intrinsic functions perform operations such as converting a value from one data type to another and perform basic mathematical

functions, such as finding sines, cosines, and square roots of numbers. The *HP FORTRAN 77/iX Reference Manual* describes each intrinsic function in detail and discusses the data types of arguments allowed and the argument and function type.

Arguments to Subprograms

Communication between the calling program and the referenced subprogram is accomplished by passing values through an argument list. An argument list consists of one or more items separated by commas and enclosed in parentheses. In addition, values can be passed through common blocks to and from subprograms.

The calling program unit passes actual arguments to a subprogram. Dummy arguments “refer” to the same locations as actual arguments; therefore, arguments are passed by reference.

Actual arguments can be variables, array names, array elements, character substrings, subprogram names, record names, record field names, constants, expressions, and alternate return specifiers. Dummy arguments can be variables, array names, and record names.

Whenever a function or subroutine is called, an association occurs between each actual argument and its corresponding dummy argument. The first actual argument is associated with the first dummy argument, the second actual argument with the second dummy argument, and so on. The number of actual arguments must be the same as the number of dummy arguments (although certain intrinsic functions allow a variable number of arguments). Also, actual arguments in a subroutine call or function reference should agree in order and data type with the corresponding dummy arguments.

To see how the dummy and actual arguments correspond, look at the following example:

```
PROGRAM main

DIMENSION q(20), r(20)

EXTERNAL fcn      ! Required for fcn to be an actual argument;
C                otherwise will be treated as a variable.
.
.
.
CALL sub1(q, x, i, r(1), fcn)
.
.
.
END

SUBROUTINE sub1(array, z, in1, tmp, f)
DIMENSION array(20)
.
.
.
r = f(in1, tmp)
END
```

The arguments correspond to each other as follows: **q** is an array name, so the dummy parameter **arry** must be dimensioned in the subprogram. **x** is a real variable; the dummy parameter (**z**) in the second position of the subroutine argument list must also be a real variable. **i** corresponds to **in1**. **r(1)** is an element of array **r** and can correspond to a single variable name (**tmp**) (not dimensioned) or an array (dimensioned) in the dummy argument list. **fcn** is a function name; therefore, **f** must be used in the context of a function in the subprogram.

A subprogram uses the actual arguments passed from the calling program to replace the dummy arguments and perform the computation. For example, consider this program:

```
PROGRAM example

INTEGER a, b
READ (5,*) a, b
WRITE(6,*) a, b

CALL switch(a,b)

WRITE(6,*) a, b
END

SUBROUTINE switch(x,y)
INTEGER x, y, temp
temp = x
x = y
y = temp

END
```

The calling program unit passes actual arguments **a** and **b** to the subroutine **switch**. The subroutine uses variables **x** and **y** as dummy arguments. Because the actual arguments are passed by reference, the variables **x** and **y** refer to the storage locations of variables **a** and **b**. The variables will then assume the current value of the actual arguments. Changing the values of the dummy arguments passed by reference changes the values of the actual arguments in the calling program unit.

Some examples of how statement functions define their arguments are shown below.

The statement:

$$\text{func}(q, r, s) = q * r / s$$

is a statement function with simple variables.

The function description:

```
FUNCTION next(z, i, j)
DOUBLE PRECISION i
```

```
DIMENSION j(10)
```

defines these arguments: **z** is a REAL variable; **i** is a REAL*8 variable; **j** is a 10-element INTEGER array.

The subroutine description:

```
SUBROUTINE add(q, f, get)
  q = get(f)
```

defines these arguments: **q** and **f** are real variables and **get** is a function name. In the context in which **get** is used, **get** could either be an array or a function. But, because **get** was not declared as an array, **get** is a function name.

All variable names are local to the program unit that defines them. In a statement function, all actual variable names are local to that statement. Similarly, dummy arguments are local to the subprogram unit or statement function containing them. Therefore, the dummy arguments can be the same as names appearing elsewhere in another program unit. No element of a dummy argument list can occur in a common (except as a common block name), EQUIVALENCE, or DATA statement.

If the actual argument is a constant, symbolic name of a constant, function reference, expression involving operators, or expression enclosed in parentheses, the associated dummy argument must not be redefined within the subprogram.

Passing Constants

FORTRAN accepts a constant value as an argument. For example, a call to a subroutine can look like this:

```
CALL sublib(books, num, 4.0)
.
.
.
SUBROUTINE sublib(titles, number, value)
```

The call to the subroutine **sublib** causes the subroutine to associate the constant 4.0 with the third dummy argument, **value**.

Because a constant cannot be changed in value, the dummy argument in the subprogram that corresponds to the actual constant should not be redefined in the subprogram.

Passing Expressions

You can use expressions as actual arguments; an actual argument can be any legal expression whose result is a value of the same data type as the dummy arguments.

For example, consider these statements:

```
CALL baseball(team + 5.0, player1, player2,  
*           SQRT(3.0 - num), win, loss)
```

```
SUBROUTINE baseball(home, member1, member2,  
*           value, wscore, lscore)
```

When the call to the subroutine `baseball` occurs, FORTRAN evaluates each expression and associates the result with the corresponding entry in the dummy argument list, as follows:

Dummy Argument	Corresponding Actual Argument
home	result of (team + 5.0)
member1	player1
member2	player2
value	result of SQRT(3.0 - num)
wscore	win
lscore	loss

You can pass character expressions and character expressions involving concatenation of operands whose lengths are (*), indicating undefined length. For example, the following program shows how to pass character expressions:

```
PROGRAM main
CHARACTER*10 string1, string2, string3, string4

string1 = 'one'
string2 = 'two'
string3 = 'three'
string4 = 'four'

CALL sub(string1, string2, string3, string4)

END

SUBROUTINE sub(a, b, c, f)
CHARACTER*(*) a, b, d, e, f
CHARACTER*20 c
PARAMETER (d = 'string1', e = 'string2')
c = a // b
```

* Legal character expressions as actual arguments:

```
CALL check(c)
CALL check(d)
CALL check(d // e)
CALL check(f)
CALL check(a // b)
CALL check(a // d)

END

SUBROUTINE check(z)
CHARACTER *(*) z

print *, z

END
```

The output from this program is as follows:

```
one      two
string1
string1string2
four
one      two
one      string1
```

When an actual argument is an expression, the dummy argument in the subprogram unit that corresponds to the actual expression should not be reassigned in the subprogram.

Passing Character Data

When character data is passed to a subprogram, both the dummy and actual arguments must be a character data type. The length of the dummy argument must be less than or equal to that of the actual argument. If the length of the dummy argument is less than the length of the corresponding actual argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are associated with the dummy argument. For example, if an actual character argument is a variable assigned the value `abcdefgh` and the length of the dummy argument is four, only the characters `abcd` are associated with the dummy argument.

Here is an example of a character argument:

```
FUNCTION size(string)
CHARACTER*10 string
```

If a dummy argument of type character is an array name, the length restriction applies to the entire array and not to each array element. The length of an individual dummy array element can be different from the length of an actual array element or array element substring. For example, a main program can have the statements:

```
FUNCTION main
CHARACTER a(10)*20    ! Length of 20 declared

CALL sub(a)
.
.
.
END
```

and the subroutine `sub` can have the following statements:

```
SUBROUTINE sub(b)
CHARACTER b(10)*10    ! Length of 10 declared
.
.
.
END
```

The length of the dummy array element `b` differs from that of the corresponding actual array element `a`.

The dummy argument array must not extend beyond the end of the associated actual argument array. For example, the program:

```
PROGRAM main
CHARACTER a(10)*10      ! 10 elements of length 10 declared

CALL sub(a)
.
.
.
END

SUBROUTINE sub(b)
CHARACTER b(20)*20     ! 20 elements of length 20 declared
.
.
.
END
```

could have unexpected results.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, the sum of the lengths of the operands is the length of the actual argument.

The length of a dummy argument can be declared by an asterisk, as shown below:

```
      SUBROUTINE sub(char_dummy)
      CHARACTER*(*) char_dummy
```

In this example, the dummy argument `char_dummy` assumes the length of the associated actual argument for each reference of the subroutine. If the actual argument is an array or array element name, the length assumed by the dummy argument is that length.

Passing Arrays

Functions and subroutines can process entire arrays. Association between an actual array argument and the corresponding dummy array argument follows the same rules described for single-valued arguments. The array name in a dummy argument list is defined as an array in a type or DIMENSION statement within the subprogram, and a similar declaration appears in the invoking program for the actual array name.

You should make sure that the declared array type is the same for both array names. For instance, if a main program has the statements:

```
PROGRAM main
INTEGER*2 a(24)
.
.
.
CALL mysub(a)
.
.
.
END
```

the subroutine mysub must include a similar declaration, like this:

```
SUBROUTINE mysub(b)
INTEGER*2 b(24)
.
.
.
END
```

If the subroutine processes an array of character strings, the declared lengths must also match.

Because each element of an array can be uniquely identified and used just like a single-valued variable, an array element can be used as an argument to a subprogram. For example, the statement:

```
CALL suba(int1, int2, 5.0, 9, array(3), array(5))
```

is a valid call to a subroutine subprogram.

Because only the name of an array appears in the dummy argument list of a subprogram, an array must be declared in a type or DIMENSION statement. The number and size of an actual argument array can differ from the number and size in the corresponding dummy argument array. The size of the dummy argument array cannot exceed the size of the actual argument array. Because array bounds across separate compilation units are not checked at run time, no warning is issued if the dummy array size exceeds the actual array size. Altering these unreserved locations could yield unpredictable results or run-time errors.

Adjustable Dimensions

Normally, array bounds are specified by integer constants and are fixed by the values of these constants. However, you can use *adjustable arrays* in subprograms. Adjustable dimensions allow you to create a subprogram that can accept varying sizes of actual array arguments. For adjustable declarations, one or more of the array bounds is specified by an expression involving integer variables or expressions, rather than by integer constants.

For example, here is a subroutine with a fixed array declared:

```
SUBROUTINE sub1(array)
  DIMENSION array(25)
```

The exact number of elements that `array` contains is 25 elements. An array that can contain a variable number of elements is declared like this:

```
SUBROUTINE sub2(array, n)
  DIMENSION array(n)
```

The value of `n` must be passed as an actual argument or must be in a common block.

An example of an adjustable array declaration in a program is shown below. The example declares an array `iarr` in the main program. The array `iarr` has two dimensions of 10 elements each. A subroutine `sb` is called to fill `iarr` with values. The variables `i` and `j` are set equal to the array bounds and these variables are used as actual arguments to be passed to the subroutine. The subroutine dummy arguments `k` and `m` assume the values passed to them through `i` and `j`. These variables are used in an `INTEGER` statement to establish the bounds for array `ivar`.

```
PROGRAM ardim

  INTEGER iarr(10, 10)
  i = 10
  j = 10
  CALL sb(iarr, i, j)
  WRITE (6, '(1X, 10I3)') iarr

END

SUBROUTINE sb(ivar, k, m)

  INTEGER ivar(k, m)
  DO nr = 1, k
    DO nc = 1, m
      ivar(nr, nc) = nr * nc
    END DO
  END DO
```

RETURN
END

The following output is produced by the program:

```
1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90100
```

Assumed-Size Arrays

The *assumed-size array* is another form of adjustable dimensions in subprograms. For assumed-size arrays, the subscript of the last dimension of the array is specified by an asterisk. Because the last bound of the dimension is not passed as an argument, the bound can take any value. The results are unpredictable when the dummy array is referenced and the last dimension index expression exceeds the size of the actual argument.

The following example demonstrates the use of assumed-size arrays. The last subscript of the array `array` can take any value from 0 to 6.

```
PROGRAM main
  DIMENSION a(10, 10)
  CALL sub1(a)
  .
  .
  .
END

SUBROUTINE sub1(z)
  DIMENSION z(10, *)
  INTEGER num(5, 10, 0:6)
  j = 5
  i = func1(num, j)
  .
  .
  .
END

FUNCTION func1(array, k)
  INTEGER array(k, 10, 0:*)
  .
  .
  .
END
```

A variable that dimensions a dummy argument in a bounds expression within a type or DIMENSION statement in a subprogram can appear either in a common block or as a dummy argument, but not both.

Adjustable and assumed-sized array declarations cannot be used in COMMON statements, in main programs, or in BLOCK DATA subprograms.

You can pass the names of functions and subroutines as arguments to other subprograms. All subprogram names used as actual arguments must be listed in an EXTERNAL statement in the calling program.

Any intrinsic function name that is an actual argument must appear in an INTRINSIC statement in the calling program. For example, the statement:

```
INTRINSIC sqrt, cos
```

specifies that the program intends to invoke one or more subprograms for which `sqrt` and `cos` are to be actual arguments used in functions. If the INTRINSIC statement is not included, `sqrt` and `cos` are assumed to be variables.

An intrinsic function name can appear in an EXTERNAL statement to allow the function name to be used as an actual argument. For example, the statement:

```
EXTERNAL sin, tan
```

specifies that the user-written subprograms `sin` and `tan` will be used as arguments. As a result, the intrinsic functions `SIN` and `TAN` cannot be used in that program or subprogram.

If the subprogram name is not listed in an EXTERNAL or INTRINSIC statement, the FORTRAN compiler treats the subprogram name as a simple variable.

Multiple Entries into Subprograms

A subroutine call or function reference usually invokes the subprogram at the entry point defined by the SUBROUTINE or FUNCTION statement. The first statement executed is the first executable statement in the subprogram. However, you can use the ENTRY statement to define other entry points within the function or subroutine.

The entry point can be anywhere within a function or subroutine after the FUNCTION or SUBROUTINE statement, but not within an IF block or DO loop. The ENTRY statement can only be used in a subroutine or function subprogram, not in a main program or block data subprogram. A subprogram can have any number of ENTRY statements.

The ENTRY statement, a nonexecutable statement, looks like this:

```
ENTRY name(argument list)
```

where **name** is the entry point name, and the optional argument list is made up of variable names, array names, dummy procedure names, or an asterisk. The asterisk, indicating an alternate return, is permitted only in a subroutine.

When an entry name is used to enter a subprogram, execution begins with the first executable statement that follows the ENTRY statement. The flow of control is illustrated in the following diagram.

```
          PROGRAM main
|<---- CALL entry1(val)
|      CALL entry2(val) ----->|
|
|      END
|
|      SUBROUTINE sub
|
|-----> ENTRY entry1(a)
|          a = a + 5.0
|          RETURN ! Return to main
|
|
|          ENTRY entry2(a) <-----
|          a = a + 10.0
|          END ! Return to main
```


A subroutine with entry points `search` and `punctuation` is shown below:

```

SUBROUTINE linka(d, n, f)
  INTEGER d, n, f, table, document

C  In subroutine linka, via primary entry point
  DO 10 i = 1, f, n
    .
    .
    .
10  CONTINUE
    RETURN

      ENTRY search(table, f)
C  In subroutine linka, via entry point search
  DO 20 i = 1, f
    .
    .
    .
20  CONTINUE
    RETURN

      ENTRY punctuation(document)
C  In subroutine linka, via entry point punctuation
  DO 30 i = 1, 5
    .
    .
    .
30  CONTINUE
    END

```

In this subroutine, the names `search` and `punctuation` define alternate entry points into subroutine `linka`.

The first statement executed in the subroutine is determined by the entry point, as follows:

Call to the Entry Point	First Statement Executed
CALL linka(var1, var2, var3)	DO 10 I = 1, f, n
CALL search(var1, var2)	DO 20 I = 1, f
CALL punctuation(var1)	DO 30 I = 1, 5

The order, type, and names of the dummy arguments in an ENTRY statement can differ from the dummy arguments in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement. Type agreement is not required for actual arguments that have no type, such as a subroutine name or an alternate return specifier as an actual argument.

The following example shows a function with entry points of different data types:

```

REAL FUNCTION f(x)      ! Real function f
INTEGER k, i
.
.
.
ENTRY k(i)              ! Integer function k
.
.
.
END

```

The declarations of the dummy arguments can precede their use in an ENTRY statement. For example, in the following function:

```

FUNCTION x(array1, q)
INTEGER q
INTEGER array1(q)
.
.
.
ENTRY y (r, q)
.
.
.
array1(q) = 100
.
.
.
END

```

the variable `q` is declared before the ENTRY statement and the last element of the array is set to 100.

In a function subprogram, a variable name that is the same as an entry name cannot appear in any statement that precedes the appearance of the entry name in an ENTRY statement, except in a type statement.

Within a subprogram, an entry name cannot appear both as an entry name in an ENTRY statement and as a dummy argument in a

FUNCTION, SUBROUTINE, or ENTRY statement. An entry name cannot appear in an EXTERNAL statement.

Common Blocks

In addition to passing values through arguments, common blocks can provide communication between program units and subprograms. Before a FORTRAN program is executed, computer storage is allocated for each program and subprogram. In addition, a common block of storage is reserved for use by all program units. Program units define the data to be reserved in common blocks with the COMMON statement.

There are two kinds of common blocks: blank common and labeled common.

Blank Common Blocks

The blank COMMON statement looks like this:

```
COMMON list
```

where `list` is variable names, array names, or array names with declared dimensions, all separated by commas. The COMMON statement instructs the compiler to establish an area of storage shared by all program units using blank COMMON statements.

For example, if the statements:

```
REAL time, distance, car(2,3)
INTEGER count
COMMON car, count, time, distance
```

are in a program unit A, during execution, the common storage is organized as follows:

Word	Item
1	car(1,1)
2	car(2,1)
3	car(1,2)
4	car(2,2)
5	car(1,3)
6	car(2,3)
7	count
8	time
9	distance

If a program unit B contains the same set of statements, each reference made to `car`, `count`, `time`, or `distance` references the same storage accessed by program unit A.

Within another program unit, the same data can be known by a different symbolic name. For example, if program unit C contains the statements:

```
REAL array1(2), array2(3)
INTEGER i, j
COMMON i, array1, j, array2
```

The common block would be accessed as follows:

Word	Item Name From Program Unit C	Item Name From Program Unit A
1	i	car(1,1)
2	array1(1)	car(2,1)
3	array1(2)	car(1,2)
4	j	car(2,2)
5	array2(1)	car(1,3)
6	array2(2)	car(2,3)
7	array2(3)	count

As you can see, inconsistent COMMON statements make a program hard to follow. To avoid errors, modules containing a COMMON statement should specify the same organization of the common area. This can be accomplished by declaring the common area in an INCLUDE file, and including the file in each subprogram that needs it.

As an extension to the ANSI standard, variables in blank common blocks can be initialized using DATA statements.

An example of how common blocks can pass values to and from subprograms is shown below. The variable `q` in the main program shares storage space with `x` in the subroutine. When a value for `q` is determined by the READ statement, `x` automatically shares this value. Similarly, `r` and `y` also share storage space, as does the variable `side` in the main program and in the subroutine. The subroutine uses the values input for `q` and `r` to compute the length of the hypotenuse of a right triangle.

```
PROGRAM comex

COMMON q, r, side

READ *, q, r
CALL tri
PRINT *, side
END

SUBROUTINE tri

COMMON x, y, side
side = SQRT(x**2 + y**2)

END
```

Labeled Common Blocks

In some programs, you might want to subdivide the common area into smaller blocks with each block having a unique name. To do this, the labeled form of the COMMON statement is used as follows:

```
COMMON /name/list, ..., /name/list
```

where **name** is the common block name and **list** is the list of variable names, arrays, and array declarators.

Before a program is executed, one block of storage is allocated for each unique named common area that was specified from the program units.

Labeled common blocks allow each program unit to have its own named common area.

For example, the statement:

```
COMMON /block1/a, b, c, /block2/x, y, z
```

defines these two labeled common blocks:

Common Block block1:

a
b
c

Common Block block2:

x
y
z

To see how labeled common blocks work, consider this partial program:

```
PROGRAM main
COMMON var1, var2, var3
COMMON /block1/ var4, var5, var6
.
.
.
END

SUBROUTINE sub(x)
COMMON /block1/ a, b, c
.
.
.
END

FUNCTION func(y)
COMMON f1, f2, f3
.
.
.
END
```

The items `var4`, `var5`, and `var6` in the main program are in the common block named `block1`. The same storage words are referred to by the names `a`, `b`, and `c` in subroutine `sub`. The items `var1`, `var2` and `var3` in the main program are in blank common. The same storage words are used by the names `f1`, `f2`, and `f3` in function `func`.

Unlike local variables in a subprogram, items in blank and named common blocks remain defined after the execution of a `RETURN` or `END` statement in a subprogram.

Arrays and variables in labeled or blank common can be initialized by using `DATA` statements in a `BLOCK DATA` subprogram.

Block Data Subprograms

Block data subprograms define the size and reserve storage space for common blocks. Block data subprograms can also initialize the variables and arrays declared in the common block. A block data subprogram begins with a BLOCK DATA statement and ends with an END statement.

For example, consider this block data subprogram:

```
BLOCK DATA datablock1
INTEGER i, n, t
REAL    x, y, z

COMMON /block1/ n, t
COMMON /block2/ i, x(10)

DATA n, t /5, 25/
DATA x /10*1.0/

END
```

This block data subprogram specifies that `n` and `t` are in named common block `block1` and these are to be initialized to 5 and 25, respectively. The block `block2` contains `i` and the ten elements in array `x`. The variable `i` is undefined and each element in array `x` is initialized to 1.0.

The block data subprogram is a nonexecutable program module that can be placed anywhere after the main program. The name of the subprogram can be omitted, but a program cannot have more than one unnamed block data subprogram.

The BLOCK DATA statement must be the first noncomment statement in a block data subprogram. Each common block referenced in an executable FORTRAN program can be defined in a block data subprogram.

Specification statements, data initialization, and blank common statements are allowed in the body of a block data subprogram. Block data statements cannot contain executable statements. Acceptable statements include: COMMON, DATA, DIMENSION, IMPLICIT, PARAMETER, SAVE, and type specification (INTEGER*4, REAL*8, and so on), statements. EXTERNAL and INTRINSIC statements are not allowed.

Here is another example of a BLOCK DATA subprogram:

```
BLOCK DATA null

COMMON /xxx/ x(5), b(10), c
COMMON /set1/ iy(10)
DATA iy/1,2,4,8,16,32,64,128,256,512/
DATA b/10*1.0/

END
```

The name `null` is the optional name of a `BLOCK DATA` subprogram used to reserve storage locations for the named common blocks `xxx` and `set1`. Arrays `iy` and `b` are initialized in the `DATA` statements. The remaining elements in the common block can optionally be initialized or typed in the block data subprogram.

Using the SAVE Statement

The SAVE statement retains the value of local variables after the execution of a RETURN or END statement in a function or subroutine. The SAVE statement allows data to be shared among subprograms because the values of entities are saved beyond the scope of the program units in which they are declared. However, an item in a common block can become undefined or redefined in another unit.

The items that can be specified by the SAVE statement are: named common blocks enclosed in slashes, a variable name, or any array name. Each item can appear only once. The items that cannot be specified by the SAVE statement are: dummy argument names, subprogram names, and names of individual items in a common block. If no individual items are specified, all variables, arrays, and common block data are saved.

If a common block name is in a SAVE statement in one subprogram of an executable program, the block name must be specified in a SAVE statement in every subprogram in which it appears. A common block name surrounded by slashes in a SAVE statement specifies all the entities in the block.

Execution of a RETURN or END statement within a subprogram causes the items in a subprogram to become undefined, except for:

- Items specified by SAVE statements.
- Items in common blocks that are not declared in the calling program.
- Items that have been defined in a DATA statement and not redefined.

The following program shows how the SAVE statement works:

```
PROGRAM main
  INTEGER sub1

  WRITE(6,*) sub1(.TRUE.), sub1(.FALSE.),
*           sub1(.FALSE.), sub1(.FALSE.)

  END

  INTEGER FUNCTION sub1(first)
  INTEGER count
  LOGICAL first
  SAVE

  IF(first) count = 0

  count = count + 1
  sub1 = count
```

```
RETURN  
END
```

The output of this program looks like this:

```
1 2 3 4
```

The variable `count` is incremented each time `sub1` is called. The `SAVE` statement in the subroutine saves the value of `count` until the next call to `sub1`.

Writing Efficient Programs

Ideally, a program should compile quickly, run quickly, and use a minimal amount of memory for code and data.

You can take steps to minimize both time and space used by a program; however, you might have to trade one type of efficiency to achieve another. For example, on a machine with 32-bit words, your program might run faster if 32-bit integers (`INTEGER*4`) are used, but will use twice as much data space as using 16-bit integers. It is up to you as to which of the resources are most valuable in your programming environment and to consider the trade-offs. Avoid optimization techniques that decrease the readability, clarity, portability, and maintainability of your program.

Note



The development of an efficient algorithm is the most important step towards efficient coding. The improvement achieved by the source manipulation techniques described in this chapter may be minimal compared to improving the overall method of solving the task.

Also, the suggestions in this chapter might reduce the readability of your program.

This chapter describes ways you can improve your program efficiency in these five areas:

- Compile time
- Run time
- Code space
- Data space
- Operating system issues

Compile-Time Efficiency

To avoid inefficient compiler options, do not use compiler options that generate symbol table information, code offsets, and code listings. These options cause the compiler to generate additional information and should be used only when the information generated is needed for debugging.

Run-Time Efficiency

The suggestions below help to decrease the time needed to run your program.

Declare Integer and Logical Variables Efficiently

Declare integer and logical sizes equal to 32 bits (4 bytes) which is the HP 3000 Series 900 word size. That is, use the defaults of INTEGER*4 and LOGICAL*4.

Avoid Using Arrays

When possible, avoid using arrays because they are addressed indirectly; that is, their address must be found first, and then the element at that address must be found.

Use Efficient Data Types

When a choice of data type is possible for a variable, choose according to the following hierarchy:

Table 6-1. Data Type Efficiency

Type of Data Type	Order of Efficiency
Floating Point	REAL*4 (fastest) REAL*8 REAL*16 (slowest)
Integer	INTEGER*4 (fastest) INTEGER*2 (slowest)
Complex	COMPLEX*8 (fastest) COMPLEX*16 (slowest)

However, as mentioned below, it is ideal to have variables that are used together in expressions to be all the same type.

Avoid Mixed-Mode Expressions

Avoid mixed-mode expressions. For example, the assignment statement:

```
int = 1.0 + int
```

where `int` is an integer, requires converting `int` to a real number and then converting the result back to an integer during execution. A more efficient assignment statement would be:

```
int = 1 + int
```

Eliminate Slow Arithmetic Operators

When possible, replace slower arithmetic operations with faster operations. The arithmetic operations are listed below from fastest to slowest:

Addition and subtraction	+ -	(Fastest)
Multiplication	*	
Division	/	
Exponentiation	**	(Slowest)

In some cases, multiplication operations can replace exponentiation, addition operations can replace multiplication, and multiplication operations can replace division. For example,

$i = j^{**2}$	can be written as	$i = j * j$
$a = 2.0 * b$	can be written as	$a = b + b$
$x = y / 10$	can be written as	$x = y * 0.1$

Note



The last example might cause an error if you are porting your program to another system because the internal representation of 0.1 might vary between systems.

Use Statement Functions

Use statement functions instead of short function subprograms. This eliminates the overhead involved with loading parameters and avoids a procedure call for the subprogram call. However, statement functions are expanded in-line and thus increase the program size.

Reduce External References

Eliminate unnecessary function calls. For example, the statement:

$$c = \log(a) + \log(b)$$

can be rewritten as:

$$c = \log(a * b)$$

Also, the statement :

$$x = y^{**2}$$

explicitly requires a procedure call to an exponential function procedure. Rewriting this statement as:

$$x = y * y$$

avoids the procedure call.

If a function is called more than once with the same arguments, you can eliminate the additional procedure calls by assigning the result to a temporary variable. For example, the statements:

```
a = MIN(x,y) + 1.0
b = MIN(x,y) + 4.0
```

can be rewritten as:

```
minval = MIN(x,y)
a = minval + 1.0
b = minval + 4.0
```

The rewritten statements above are more efficient, but the readability is reduced.

Combine DO Loops

Combine adjacent DO loops that are executed the same number of times. For example, the statements:

```
DO 100 i = 1,20
100   a(i) = b(i) + c(i)
DO 200 j = 1,20
200   x(j) = y(j) + z(j)
```

can be replaced with the statements:

```
DO 100 i = 1,20
   a(i) = b(i) + c(i)
   x(i) = y(i) + z(i)
100 CONTINUE
```

Eliminate Short DO Loops

Break short DO loops into separate statements to eliminate the overhead associated with the loop. For example, the statements:

```
DO 50 i = 1,3
50   c(i) = a(i) * b(i)
```

can be replaced with the statements:

```
c(1) = a(1) * b(1)
c(2) = a(2) * b(2)
c(3) = a(3) * b(3)
```

However, removing DO loops makes programs longer and is not practical if the number of loop iterations is large.

Eliminate Common Operations in Loops

Minimize operations inside of a loop. If the result of an operation is the same throughout the loop, move the expression before or after the loops so the expression is only executed once. For example, the loop:

```
        sum = 0.0
    DO 100 i = 1,n
100      sum = sum + value * a(i)
```

can be replaced with:

```
        sum=0.0
    DO 100 i = 1,n
100      sum = sum + a(i)
        sum = value * sum
```

Use Efficient IF Statements

In block IF statements, order the conditions so that the most likely condition is tested first. For example, if the value of `arg` is three in most cases, write a compound IF statement as:

```
    IF (arg .EQ. 3) THEN
        .
        .
        .
    ELSE IF (arg .EQ. 1) THEN
        .
        .
        .
    ELSE IF (arg .EQ. 2) THEN
        .
        .
        .
    ELSE
        .
        .
        .
```

When using the logical operators `.AND.` and `.OR.` in an IF condition, the code generated only checks enough conditions to determine the result of the entire logical expression. If several logical expressions are connected with `.OR.`, checking discontinues as soon as an expression evaluates to `.TRUE.`. When `.AND.` is used, checking discontinues as soon as a `.FALSE.` condition is found. Therefore, order the conditions so the least number of checks is done. For example, if it is more likely that variable `a` will equal zero than it is that `b` will be greater than 100, write the IF statement as:

```
    IF ((a .EQ. 0) .OR. (b .GT. 100))
```

or:

```
    IF ((b .GT. 100) .AND. (a .EQ. 0))
```

Avoid Formatted I/O

When possible, use unformatted I/O. Formatted I/O requires costly conversions between binary and ASCII format.

When using formatted I/O, put the format string in a separate FORMAT statement instead of using a variable. For example, use the statements:

```
        WRITE (6,20) var
20     FORMAT (F10.2)
```

instead of:

```
CHARACTER*7 a
DATA a/'(F10.2)'/
WRITE (6,a) var
```

Format specifiers contained in variables are not parsed when a program is compiled. Instead, a format processing routine is called by the compiled program each time the format is used.

Specify the Array Name for I/O

When reading or writing an array, specify the array name instead of using an implied DO loop. This allows the array to be operated on as a whole, instead of performing individual operations for each element. For example, specify:

```
WRITE (6, '(A1)') myarray
```

instead of:

```
WRITE (6, '(A1)') (myarray(i,j), i=1,10), j=1,10)
```

Avoid Using Range Checking

Turn range checking on only when necessary. This option causes extra code to be included in your program to check the bounds when a substring or array element is referenced. Code is also generated for checking assigned GOTO statements. This added code causes your program to take longer to execute, as well as using additional code space.

Use Your System Language

In some cases it might help efficiency to write part of your program in the system language of your machine. For example, if you have to move an entire array to another array with the same dimensions, your system language might allow you to move the array as a single block instead of moving each array element separately. If the array is large, using the system language could save a significant amount of execution time.

Minimize Segment Faults

On systems using memory segmenting, segment your program with efficiency in mind. If a large amount of interaction takes place between two program units, make sure that the program units are placed in the same segment. Try to minimize the total number of segments used, without making any one segment too large.

MPE/iX Run-Time Efficiency Topics

To improve run-time efficiency on the MPE/iX operating system, do the following:

- When using the CHECK_OVERFLOW compiler directive, specify

```
$CHECK_OVERFLOW INTEGER OFF
```

If left in the default state of ON, the directive generates extra code for each integer assignment for integer overflow checking.

- Use MPE/iX intrinsic I/O instead of the FORTRAN READ and WRITE statements. The FORTRAN statements generate several procedure calls for each statement. However, MPE/iX intrinsics make programs system-dependent and difficult to port.
- When possible, use DO loops instead of DO WHILE loops. The code generated to evaluate the DO loop counter is more efficient if the loop counter is type INTEGER*4 and if there are no ASSIGN statements in the program unit.
- Avoid using common variables, variables initialized by DATA statements, arrays, equivalenced data, and variables with a length greater than 64 bits. These structures are addressed indirectly; that is, their addresses must first be found and then the elements at those addresses found.
- Avoid using the HP3000_16 compiler directive. Instead, change equivalence and common data to take advantage of the HP Precision Architecture, convert files that contain real data to IEEE format, and modify character assignments if your application takes advantage of the ripple effect of overlapping character strings.
- Use the LOCALITY compiler directive to strategically place subroutines and functions in memory.

Code Space Efficiency

The suggestions below help to decrease the amount of space needed to store your program.

Use Function Subroutines

Use function subprograms instead of statement functions because the code to execute statement functions is generated every place the function is called. Of course, you are sacrificing run-time efficiency because an extra procedure call is executed for the subprogram call.

Avoid Formatted I/O

When possible, avoid formatted I/O because the format strings are stored in your program.

Use Character Substrings

When assigning a character constant to a character variable that is longer than the constant, specify a substring equal to the constant size for the variable. For example, if `charstring` is longer than three characters, use the statement:

```
charstring(1:3) = 'ABC'
```

instead of:

```
charstring = 'ABC'
```

This eliminates the code that is generated to fill the remainder of the character variable with blanks. However, this should be done only if you do not need the remainder of the variable to be filled with blanks.

Data Space Efficiency

The suggestions below help to decrease the amount of space needed to store data.

Eliminate Redundant or Unused Variables

Eliminate redundant or unused variables. Redundant variables are defined and used only one time. For example, in the statements:

```
temp1 = a * 25.0
temp2 = b**3
answer = temp1 + temp2
```

the variables `temp1` and `temp2` are redundant. The three lines can be rewritten as:

```
answer = a * 25.0 + b**3
```

The cross reference facility can be used to locate the redundant and unused variables. Refer to the `CROSSREF` or `XREF` compiler directive in the *HP FORTRAN 77/iX Reference*.

Avoid Common Variables

Common variables remain in your data space throughout the run of your program, but local variables typically are only in the data space when the subroutine in which they are declared is active. Do not place variables that are accessed by only one routine into common blocks.

When using `COMMON` and `EQUIVALENCE` statements, group variables of the same type. This prevents wasted space due to differences in data type alignment.

Use `INTEGER*2` and `LOGICAL*2` Data

To save data space, use the `SHORT` compiler option so that integer and logical data use 16 bits instead of 32 bits. However, be aware that some constructs require 4-byte types. Among these are all integer and logical I/O specifiers whose values are set by the I/O library (such as most `INQUIRE` statement specifiers) and integer variables in `ASSIGN` statements.

Note



You can decrease performance by using 16-bit data.

Performance Tuning

This section provides more information on how your program interacts with the MPE/iX operating system on the 900 HP Series 3000 computer and what can be done to optimize the relationship. Before reading this chapter, you should understand the memory model used by the compiler.

Grouping Related Routines

To understand how a program uses memory, you need to know how these areas are mapped into the physical segments of the 900 Series HP 3000 computer.

The main memory is divided into page frames of a fixed size, each of which can contain one virtual page. Data flow between main memory and external storage occurs in units of pages; the page size is 4K bytes. The stack and data areas belong to a separate virtual space from the code area. The maximum size of each virtual space is 2^{32} bytes, but the code, data, and stack areas only use one-fourth of the space. Therefore, the maximum size of the code area is 2^{30} bytes and the maximum size of the stack and data area combined is 2^{30} bytes.

You can make programs more efficient by grouping routines that frequently call each other. Fitting routines into a page boundary eliminates page faults that might otherwise occur. (Grouping data that is frequently used also lessens the possibility of a page fault.) Group routines using the LOCALITY compiler directive (refer to the *HP FORTRAN 77/iX Reference* for more information about this directive). When you use the LOCALITY directive in routines with the same locality name, the compiler arranges the routines next to each other in the code space. Specifying a locality name for block data subprograms that contain common blocks also causes the compiler to place common blocks having the same locality name next to each other. Therefore, if the common blocks fit into one page, all page faults are eliminated when accessing data within the blocks.

Besides limiting page size, the following are additional reasons to group related routines:

1. On the 900 Series HP 3000 computer, which is equipped with a cache memory, careful segmentation of code and data eliminates unnecessary cache misses. Because one cache miss is equivalent to at least two instruction cycles, a significant improvement is achieved by rearranging code and data. * Grouping related routines or data avoids long branching. On the 900 Series HP 3000 computer, the maximum distance of a branch is 64K instructions. If a routine calls another routine that is further than 64K instructions away, a *long branch stub* is generated within the 64K limit. The long branch stub serves as a transit routine, which in turn calls the target routine.

If subroutines or functions are randomly scattered in a large program, long branch stubs are likely to be generated. However, by grouping related routines with the LOCALITY directive, you can eliminate most long branchings. * The 900 Series HP 3000 computer references data through the **dp** (data pointer) and **sp** (stack pointer) base registers. The range that the computer can reach is from **dp-8K** to **dp+8K** and from **sp-8K** to **sp+8K**, respectively. If the data used by the current program is larger than 16K, the compiler has to switch the **dp** or **sp** base registers

to access data in different 16K segments. For example, if **a** is in location 0 while **b** is in location 16K+1 and the code states

```
DO i = 1,10
  a(i) = b(i)
```

the compiler must switch **dp** to point to 16K to load the value of **b**. The compiler then must switch back to 0 to store to **a(i)**. Grouping **a** and **b** could place **b** within the 16K range and eliminate the base register switching.

Shifting Data from the Data Area to the Stack Area

Another improvement in performance can be gained by shifting data from the data area to the stack area. In general, accessing data from the stack or code area takes one machine instruction, while accessing data from the data area requires two instructions. To shift data, use assignment statements if you do not require the data values to remain the same across routines. Shifting data eliminates initializing data with DATA statements. Then, instead of routines communicating through common blocks, you can use parameters.

Shifting data from the data to the stack area also produces more opportunities for the optimizer to allocate register variables, and thereby improving optimization.

Integer Overflow Checking

Integer overflow checking is turned on for 16-bit and 32-bit arithmetic to be compatible with FORTRAN 77/V. The 16-bit integer overflow checking generates three extra instructions to check the overflow for loads and stores. The 32-bit overflow checking generates overflow instructions that cannot be optimized. To turn off integer overflow checking, specify `$CHECK_OVERFLOW INTEGER OFF`.

Using the Optimizer

The optimizer is an optional part of the compiler that modifies your program so that machine resources are used more efficiently, using less space and running faster. The optimizer consists of 12 modules: five for level one optimization and seven for level two optimization.

This section describes the following:

- When and how to use the optimizer
- Level one and two optimization modules
- Optimizer assumptions
- How to write code that is easily optimized
- What to do if your optimized program fails

Introduction to the Optimizer

You can run the optimizer in one of three ways:

1. **No optimization** (this is the default).
2. **Level one optimization:** This level only performs a subset of the available optimization modules. The transformations performed are local to small subsections of code, and therefore are performed quickly with little run-time storage required by the compiler. Level one optimization should be used when some optimization is desired, but when compile time performance is more important than run time performance.
3. **Level two optimization:** This level of optimization performs all of the available optimization modules. Transformations are performed over the scope of each procedure. If you use this level of optimization, the compiler uses more memory and takes longer to process your program.

When to Use the Optimizer

Use the optimizer only on debugged code that is ready to run, because the compiler cannot generate debug information *and* perform optimizations at the same time. After level two optimizations are performed, the code is radically reordered and variable values might not be maintained in memory, which makes symbolic debugging impossible. Therefore, once a program is optimized, you cannot use symbolic debugging unless you recompile without optimization.

Invoking the Optimizer on MPE/iX

Invoke the optimizer by specifying the OPTIMIZE compiler directive in your source file or by passing the directives through the INFO string. For level one optimization, use the command

```
$OPTIMIZE LEVEL1
```

For level two optimization, use the command

```
$OPTIMIZE
```

or

```
$OPTIMIZE LEVEL2
```

Level One Optimization Modules

The level one optimization modules are:

- Branch optimization
- Dead code elimination
- Faster register allocation
- Instruction scheduler
- Peephole optimization

The examples in this section are shown at the source code level wherever possible; transformations that cannot be shown at the source level are shown in assembly language.

Table 6-2 summarizes the assembly language routines.

Table 6-2. Descriptions of Assembly Language Routines

Instruction	Description
LDW <i>offset(sr, base), target</i>	Loads a word from memory into register <i>target</i> . <i>sr</i> is the space register (0 through 7); <i>base</i> is the base register (0 through 31).
ADDI <i>const, reg, target</i>	Adds the constant <i>const</i> to the contents of register <i>reg</i> and puts the result in register <i>target</i> .
LDI <i>const, target</i>	Loads the constant <i>const</i> into register <i>target</i> .
AND <i>reg1, reg2, target</i>	Performs a bitwise AND of the contents of registers <i>reg1</i> and <i>reg2</i> and puts the result in register <i>target</i> .
COMIB , <i>cond, const, reg, lab</i>	Compares the constant <i>const</i> to the contents of register <i>reg</i> and branches to label <i>lab</i> if the condition <i>cond</i> is true.
BB , <i>cond, reg, num, lab</i>	Tests the bit number <i>num</i> in the contents of register <i>reg</i> and branches to label <i>lab</i> if the condition <i>cond</i> is true.
COPY <i>reg, target</i>	Copies the contents of register <i>reg</i> to register <i>target</i> .
STW <i>reg, offset(sr, base)</i>	Stores the word in register <i>reg</i> to memory. <i>sr</i> is the space register (0 through 7); <i>base</i> is the base register (0 through 31).

Branch Optimization Module

The branch optimization module makes branch instruction sequences more efficient whenever possible. Examples of possible transformations are:

- Deleting branches whose target is the fall-through instruction (that is, the target is two instructions away)
- When the target of a branch is an unconditional branch, changing the target of the first branch to be the target of the second unconditional branch
- Transforming an unconditional branch at the bottom of a loop, which branches to a conditional branch at the top of the loop, into a conditional branch at the bottom of the loop
- Changing an unconditional branch to the exit of a procedure into an exit sequence where possible
- Changing conditional or unconditional branch instructions that branch over a single instruction into a conditional nullification in the previous instruction
- Looking for conditional branches over unconditional branches, inverting the sense of the first branch and deleting the second branch. These result from null THEN clauses and from THEN clauses that only contain GOTO statements. For example, the code

```
IF (x) THEN
    statement 1
ELSE
    GOTO 100
ENDIF
statement 2
100 statement 3
```

becomes

```
IF (.NOT. x) GOTO 100
statement 1
statement 2
100 statement 3
```

Dead Code Elimination Module

The dead code elimination module removes unreachable code that is never executed.

For example, the code

```
if (.FALSE.) then
    a=1
else
    a=2
endif
```

becomes

```
a=2
```

Faster Register Allocation Module

The faster register allocation module, used with unoptimized code, analyzes register use faster than the advanced register allocator (a level two module).

This module performs the following:

- Inserts entry and exit code
- Generates code for operations (such as multiplication and division)
- Eliminates unnecessary copy instructions
- Allocates actual registers to the dummy registers in instructions

Instruction Scheduler Module

The instruction scheduler module performs the following:

- Reorders the instructions in a basic block to improve memory pipelining. (For example, where possible, a load instruction is separated from the use of the loaded register.)
- Where possible, follows a branch instruction with an instruction that can be executed as the branch occurs.
- Schedules floating point instructions.

For example, the code

```
LDW    -52(0,30),r1
ADDI   3,r1,r31    ;interlock with load of r1
LDI    10,r19
```

becomes

```
LDW    -52(0,sp),r1
LDI    10,r19
ADDI   3,r1,r31    ;use of r1 is now separated from load
```

Peephole Optimization Module

The peephole optimization module is a machine-dependent module that makes a pass through an intermediate representation of the code applying patterns to a small window of code looking for optimization opportunities. The optimizations performed are:

- Changing the addressing mode of instructions so they use shorter sequences
- Substituting sequences of instructions that access bit fields with shorter, equivalent instructions

For example, the code

```
LDI    32,r3
AND    r1,r3,r2
COMIB,= 0,r2,L1
```

becomes

```
BB,>=  r1, 26, L1
```

Level Two Optimization Modules

The level two optimization modules are:

- Advanced register allocation
- Induction variables and strength reduction
- Common subexpression elimination
- Constant folding
- Loop invariant code motion
- Store/Copy optimization
- Unused definition elimination

The examples in this section are shown at the source code level wherever possible; transformations that cannot be shown at the source level are shown in assembly language. See Table 6-2 for a description of the assembly language routines.

Advanced Register Allocation Module

The advanced register allocation module performs some copy optimizations, as well as allocating registers. Before the register allocator is run, the instructions contain register numbers that do not correspond to actual registers. The register allocator assigns real registers to these instructions and removes unnecessary COPY instructions.

For example, the following code shows the type of optimization the coloring register allocation module performs. The code

```
LDI    2,r104
COPY   r104,r103
LDO    5(r103),r106
COPY   r106,r105
LDO    10(r105),r107
```

becomes

```
LDI    2,r25
LDO    5(r25),r26
LDO    10(r26),r31
```

Strength Reduction

The induction variables and strength reduction module removes linear functions of a loop counter and replaces them with the loop counter. Variables of the same linear function are computed only once. This module also simplifies the function by replacing multiplication instructions with addition instructions wherever possible.

For example, the code

```
DO i = 1,10
  j(i) = i*k
END DO
```

becomes

```
t1 = k
DO i = 1,10
  j(i) = t1
  t1 = t1+k
END DO
```


Common Subexpression Elimination

The common subexpression elimination module identifies expressions that appear more than once and have the same result, computes the result, and substitutes the result for each occurrence of the expression. The types of subexpressions include instructions that load values from memory, as well as arithmetic evaluation.

For example, the code

```
a = x + y + z
b = x + y + w
```

becomes

```
t1 = x + y
a = t1 + z
b = t1 + w
```

Constant Folding Module

While the optimizer is collecting information about uses and definitions of resources, the constant folding module replaces constant expressions with their values.

For example, the code

```
a = 1
b = 2
c = a + b
```

becomes

```
a = 1
b = 2
c = 3
```

Loop Invariant Code Motion Module

The loop invariant code motion module recognizes instructions inside a loop whose results do not change and moves the instructions outside the loop.

For example, the code

```
x = z
DO i = 1,10
    a(i) = 4 * x + i
END DO
```

becomes

```
x = z
t1 = 4 * x
DO i = 1,10
    a(i) = t1 + i
END DO
```

Store/Copy Optimization Module

Where possible, the store/copy optimization module substitutes registers for memory locations by replacing store instructions with copy instructions and deleting load instructions.

For example, the following FORTRAN 77 code

```
INTEGER FUNCTION i
      .
      .
      .
      i = j + 23
      RETURN
      END
```

produces this code for the unoptimized case

```
LD0    23(r26),r1
STW    r1,-52(0,sp)
LDW    -52(0,sp),ret0
```

and this code for the optimized case:

```
LD0    23(r26),ret0
```

Unused Definition Elimination Module

The unused definition elimination module removes unused memory location and register definitions. These definitions are often a result of transformations made by other optimization modules.

For example, the function

```
INTEGER FUNCTION f(x)
INTEGER x,a,b
a = 1
b = 2
f = x * b
RETURN
END
```

becomes

```
INTEGER FUNCTION f(x)
INTEGER x,a,b
b = 2
f = x * b
RETURN
END
```

Optimizer Guidelines

These guidelines will help you use the optimizer effectively and write efficient HP FORTRAN 77 programs.

1. Where possible, expand procedures with fewer than five lines in the program or convert them to macros. The optimizer makes better use of register variables if the procedures have fewer than 100 lines. If a loop only contains a procedure call, it is more efficient to put the loop in the procedure.
2. Make hash table sizes and field sizes of variables in powers of two.
3. Where possible, use local variables to help the optimizer promote variables to registers.
4. Where possible, construct loops so the control variable increases or decreases towards zero. The code generated for a test of a loop termination is more efficient with a test against zero than for a test against some other value.
5. Where possible, use constants instead of variables for shift, multiplication, division, and remaindering.
6. Where possible, avoid using extensive equivalencing and memory mapping schemes.

HP FORTRAN 77 Optimizer Assumptions

During optimization, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program (at least to the extent that the original unoptimized program is correct). When gathering this information, the HP FORTRAN 77 compiler assumes that inside a routine (either a function or a subroutine), the only variables that can be accessed directly or indirectly or by another function call are:

- Common variables declared in this routine
- Local variables (all static variables and nonstatic variables)
- Parameters to this routine

In general, you do not need to be concerned about this assumption. Good programming practices preclude code that violates the assumption. However, if you have code that violates the assumption, the optimizer can change the behavior of the program in an undesired way. In particular, you should avoid the following coding practices to ensure correct program execution for optimized code:

- Avoid referencing outside the bounds of an array.
- Avoid using variables that can be accessed by a process other than the program, such as shared common variables. The compiler assumes that the program is the only process accessing its data. The only exception to this is if a semaphore in the form of a

function call is used to “lock” and “unlock” access to a shared variable. In this case, optimization is assumed to be correct.

OPTIMIZE Compiler Directive

The OPTIMIZE compiler directive gives you the ability to give information about the program to the compiler.

The OPTIMIZE directive controls which functions are optimized and which set of optimizations is performed. Some directives must be placed before the function to be optimized, while others can appear anywhere within the function.

This is the syntax of the OPTIMIZE compiler directive:

Syntax

```

$OPTIMIZE [
    LEVEL1
    LEVEL2
    LEVEL2_MIN
    LEVEL2_MAX
    ASSUME_NO_EXTERNAL_PARMS
    ASSUME_NO_FLOATING_INVARIANT
    ASSUME_NO_PARAMETER_OVERLAPS
    ASSUME_NO_SHARED_COMMON_PARMS
    ASSUME_NO_SIDE_EFFECTS
    ASSUME_PARM_TYPES_MATCHED
    LOOP_UNROLL [ COPIES=n SIZE=n STATISTICS ]
]
[ ON
  OFF ]
```

ON	Alone, specifies level 2 optimization. With a preceding option, sets that option off.
OFF	Alone, specifies level 0 optimization. This is the default. With a preceding option, sets that option off.
LEVEL1	Specifies level 1 optimization.
LEVEL2	Specifies level 2 optimization, with the following ASSUME settings:

ASSUME_NO_EXTERNAL_PARMS	ON
ASSUME_NO_FLOATING_INVARIANT	ON
ASSUME_NO_PARAMETER_OVERLAPS	ON
ASSUME_NO_SHARED_COMMON_PARMS	ON
ASSUME_NO_SIDE_EFFECTS	OFF
ASSUME_PARM_TYPES_MATCHED	ON
LOOP_UNROLL	ON

LEVEL2_MIN	Specifies level 2 optimization with all the ASSUME settings OFF.
LEVEL2_MAX	Specifies level 2 optimization with all the ASSUME settings ON.
ASSUME_NO_EXTERNAL_PARMs	Assumes that none of the parameters passed to the current procedure are from an external space, that is, different from the user's own data space. Parameters can come from another space if they come from operating system space or if they are in a space shared by other users.
ASSUME_NO_FLOATING_INVARIANT	Assumes that no floating invariant operations are executed conditionally with loops.
ASSUME_NO_PARAMETER_OVERLAPS	Assumes that no actual parameters passed to a procedure overlap each other.
ASSUME_NO_SHARED_COMMON_PARMs	This directive should be ON when all of the following are true: <ul style="list-style-type: none"> ■ The parameter passed to the current procedure is part of a common block used by that procedure. ■ The parameter is named differently than the variable name it has in the common block. ■ The parameter is reassigned with the same value within the procedure.
ASSUME_NO_SIDE_EFFECTS	Assumes that the current procedure changes only local variables. It does not change any variables in COMMON, nor does it change parameters.
ASSUME_PARM_TYPES_MATCHED	Assumes that all of the actual parameters passed were the type expected by this subroutine.
LOOP_UNROLL	Unrolls DO loops having 60 or less operations four times. For further details, see "Loop Unrolling" in this chapter. The default is ON.

There are five levels of optimization:

- Level 0** Does no optimizing. This is obtained by specifying `$OPTIMIZE OFF`.
- Level 1** Optimizes only within each basic block. This is obtained by specifying `$OPTIMIZE LEVEL1 ON`.
- Level 2 minimum** Optimizes within each procedure with no assumptions on interactions of procedures. That is, the compiler assumes nothing, making this the most conservative level 2 optimization. This level is obtained by specifying `$OPTIMIZE LEVEL2_MIN ON` within each procedure.
- Level 2 normal** Optimizes within each procedure with normal assumptions on interactions of procedures set as described earlier. In general, these settings are appropriate for most FORTRAN programs. This level is obtained by specifying `$OPTIMIZE LEVEL2 ON`, `$OPTIMIZE ON` or just `$OPTIMIZE` within each procedure.
- Level 2 maximum** Optimizes within each procedure with all assumptions on interactions of procedures set to OFF. This is obtained by specifying `$OPTIMIZE LEVEL2_MAX ON` within each procedure.

A basic block is a set of instructions to be executed in sequence, with one entrance, the first instruction, and one exit, the last; the block contains no branches.

Parameters can come from another space if they come from the operating system or if they are in a space shared by other users.

With level two optimization, the compiler and optimizer can achieve very sophisticated optimization. Use the ASSUME options to provide the information required for level two optimization.

`ASSUME_NO_PARAMETER_OVERLAPS` tells the compiler that the parameters passed to the current routine never overlap each other, as in the following code:

```
subroutine a(i,j,k)
.
.
.
PROGRAM b
CALL a(1,m,n)
END
```

The `ASSUME_NO_PARAMETER_OVERLAPS` option should usually be set to `ON`. However, for the following code

```
subroutine a(i,j,k)
.
.
.
END
PROGRAM b
CALL a(1,1,m)
END
```

the `ASSUME_NO_PARAMETER_OVERLAPS` option should *not* be set to `ON` because the first two parameters passed to `A` are actually the same variable (that is, the parameters overlap).

`ASSUME_NO_SIDE_EFFECTS ON` tells the compiler that all the procedure calls after this option do not change any of the common variables or the contents of the parameters being passed. For example, in the following code

```
PROGRAM a
COMMON c,d,e
$OPTIMIZE ASSUME_NO_SIDE_EFFECTS ON
CALL s1(i,j,k)
CALL s2(l,m,n)
END
```

the compiler assumes that subroutines `s1` and `s2` will not change the values of parameters `i`, `j`, `k`, `l`, `m`, `n`, or common variables `c`, `d`, or `e`.

`ASSUME_NO_PARM_TYPES_MATCHED ON` tells the compiler that the type declaration of each of the parameters in the called routine is the same as that of the caller. For example, in the following code

```
PROGRAM a
INTEGER i,j,k
$OPTIMIZE ASSUME_PARM_TYPES_MATCHED ON
CALL s1(j)
END

SUBROUTINE s1(j)
INTEGER j
j = 1
END
```

the type declaration of parameter `i` in the called routine is integer, matching the type declaration in the caller, `PROGRAM a`.

However, for the following code

```
PROGRAM a
  INTEGER i,j,k
  CALL s1(i)
END

SUBROUTINE s1(j)
  INTEGER j(3)
  j(1) = 1
END
```

the `ASSUME_PARAMETERS_MATCHED` option has to be set to `OFF` before the call to `s1` because the called subroutine `s1` declares parameter `j` to be an integer array, which is not the same as an integer in the caller `a`. Notice that `s1` is actually intended to change the contents of `j(2)`, which is not `i` but the variable following `i`. However, what follows `i` is system dependent.

`ASSUME_NO_EXTERNAL_PARMS ON` tells the compiler that none of the parameters passed to the current procedure are from an external space. That is, none are different from the user's own data space. For example, if you are accessing data in the operating system, you are accessing data from an external space. Shared data or shared common variables fall into this category. If `ASSUME_NO_EXTERNAL_PARMS` is `OFF`, the compiler is unable to perform certain optimizations, such as array accessing optimization.

The following options are meaningful only when the compiler is performing level 2 optimization, that is, only if the option `ON`, `LEVEL2`, `LEVEL2_MIN`, or `LEVEL2_MAX` has been specified:

```
ASSUME_NO_PARAMETER_OVERLAPS
ASSUME_NO_SIDE_EFFECTS
ASSUME_PARM_TYPES_MATCHED
ASSUME_NO_EXTERNAL_PARMS
ASSUME_NO_FLOATING_INVARIANT
LOOP_UNROLL
```

Default	Off.
Location	The following OPTIMIZE options must appear before any nondirective statements in the program unit: <pre> OFF ON LEVEL1 LEVEL2 LEVEL2_MIN LEVEL2_MAX ASSUME_NO_PARAMETER_OVERLAPS ASSUME_NO_EXTERNAL_PARMS ASSUME_NO_SHARED_COMMON_PARMS ASSUME_NO_FLOATING_INVARIANT </pre> These options can appear anywhere within a program unit: <pre> ASSUME_NO_SIDE_EFFECTS ASSUME_PARM_TYPES_MATCHED LOOP_UNROLL </pre>
Toggling/ Duration	The optimize options remain in effect until they are changed by another OPTIMIZE directive.
Impact on Performance	This directive can improve performance. Loop unrolling, which usually improves performance, can occasionally degrade performance because of large loops (register spilling) and code expansion (crossing the page boundary causing cache misses and TLB misses.)

Flagging Uninitialized Variables

When the compiler is performing level 2 optimization, it will detect any uninitialized non-static simple local variables. However, it will not detect uninitialized common variables, static variables, or variables of character and complex type. For example:

```

$OPTIMIZE
  FUNCTION func(type)
  COMMON /a/comvar
  SAVE statvar
  REAL foo,type
  type = 10.2
  foo = comvar
  foo = statvar
  foo = typo
  RETURN
  END

```

The variable `typo` is flagged as an uninitialized variable because it was typed incorrectly and, therefore, not initialized. However, `statvar` and `comvar` are not flagged because of their global and

static characteristics. A warning message will be issued when an uninitialized variable is detected.

Example

```
C      Start with minimum level 2 optimization.
$OPTIMIZE LEVEL2_MIN

      PROGRAM FEQ7
      INTEGER num(10), ans, calculate
      CHARACTER*2 option(10)

C
C      For the next two calls, the parameter type declarations are the same in
C      the main program and the subroutine or function.  Therefore, we can
C      further optimize the program by setting the following optimizer option.
C
$OPTIMIZE ASSUME_PARM_TYPES_MATCHED ON
      call getnum_option(num,option)

C
C      For the next call, the function will not change the parameter value or
C      any global variables in COMMON blocks.  Therefore, we can further
C      optimize the program by setting the following optimizer option.
C
$OPTIMIZE ASSUME_NO_SIDE_EFFECTS ON
      ans= calculate(num,option)
$OPTIMIZE ASSUME_NO_SIDE_EFFECTS OFF
      WRITE(6,*) 'Result = ',ans
      END

C
C      For the next subroutine, you know that the actual parameters passed
C      to this subroutine are not overlapped with each
C      other or from a space different from your program.
C      Thus, you can further optimize
C      the program by setting the following optimizer options.
C
$OPTIMIZE ASSUME_NO_PARAMETER_OVERLAPS ON
$OPTIMIZE ASSUME_NO_EXTERNAL_PARMS ON
      SUBROUTINE getnum_option(value,operation)
      INTEGER value(10)
      CHARACTER*2 operation(10)

      DO 10  i = 1,10
20      WRITE(6,*) 'Please input operation type and integer value :'
          READ(5,*) operation(i),value(i)

          IF (operation(i).EQ.' ') GOTO 30

          IF ((operation(i).NE.'**').AND.
/           (operation(i).NE.'*' ).AND.
/           (operation(i).NE.'/' ).AND.
/           (operation(i).NE.'-' ).AND.
/           (operation(i).NE.'+' )) GOTO 20
10      CONTINUE
30      RETURN
```

```

        END
C
C   For the next subroutine, you know that the actual parameters passed to
C   this subroutine are not overlapped with each
C   other and not from an external space.
C   Thus, you can leave the
C   ASSUME_NO_PARAMETER_OVERLAPS and ASSUME_NO_EXTERNAL_PARMS
C   settings ON.
C

FUNCTION calculate(value,operation)
INTEGER value(10),calculate,ans
CHARACTER*2 operation(10)

ans = 0
DO 10 i = 1,10
IF (operation(i).EQ.' ') GOTO 30

IF (operation(i).EQ.'**') THEN
    ans = ans ** value(i)
ELSE IF (operation(i).EQ.'*' ) THEN
    ans = ans * value(i)
ELSE IF (operation(i).EQ.'/' ) THEN
    ans = ans / value(i)
ELSE IF (operation(i).EQ.'-' ) THEN
    ans = ans - value(i)
ELSE IF (operation(i).EQ.'+' ) THEN
    ans = ans + value(i)
ENDIF
10 CONTINUE
30 calculate = ans
RETURN
END

```

The ASSUME_NO_FLOATING_INVARIANT option should be set to ON unless you need to turn it off for a specific subprogram. The following example illustrates this option.

```
C This program gets a divide by zero trap when compiled without
C ASSUME_NO_FLOATING_INVARIANT ON specified (the default). Because
C b/a is an invariant floating point operation (FLOP), it is moved
C out of the loop and executed whether the condition i.GT.10 is
C true or false. The ASSUME_NO_FLOATING_INVARIANT directive tells
C the optimizer not to perform this code for FLOPs that are executed
C conditionally.
```

```
$OPTIMIZE ASSUME_NO_FLOATING_INVARIANT OFF
PROGRAM test
REAL a, b, c
DATA c/1.0/, b/1.0/, a/0.0/
READ *, n

DO i=1,n
  IF (i .GT. 10) THEN
    c = b/a
  ENDIF

  c = c + i
END DO

PRINT *, a, b, c
END
```

Loop Unrolling

```
$OPTIMIZE LOOP_UNROLL [ ON  
                        OFF  
                        COPIES = n  
                        SIZE = n  
                        STATISTICS ]
```

ON	Turns on loop unrolling. ON is the default at level 2.
OFF	Turns off loop unrolling.
COPIES = <i>n</i>	Tells the compiler to unroll the loop <i>n</i> times. The default is four times.
SIZE = <i>n</i>	Tells the compiler to unroll the loops that have less than <i>n</i> operations. The default is 60 operations.
STATISTICS	Tells the compiler to give statistics about the unrolled loops.

Limits on Use

DO loops at level 2 are unrolled four times by default. If the loop limit is either not known at compile time or is less than four times, an extra copy of the DO loop body is generated. This is called unrolling the loop four or more times.

Although loop unrolling optimization usually increases performance, it can occasionally degrade performance because of large loops (register spilling) and code expansion (crossing the page boundary causing cache misses and TLB misses.) When you encounter these circumstances, you can turn off loop unrolling locally by using the compiler directive. Use the compiler directive \$OPTIMIZE to specify optimization level in the source and for changing the assumptions made by the compiler. You can use a suboption LOOP_UNROLL to control some constraints:

```
$OPTIMIZE LOOP_UNROLL
```

You can also use the `LOOP_UNROLL` suboption on the `$OPTIMIZE` directive to change the `DO LOOP` constraints for unrolling dynamically:

- You can unroll a `DO` loop more than four times.
- You can force a `DO` loop to unroll despite its large size.
- You can find the reason why a `DO` loop is not unrolled.

The highest level of optimization must be on for `LOOP_UNROLL` to work. Otherwise, `LOOP_UNROLL` is ignored. If `LOOP_UNROLL` is ignored, but `STATISTICS` has been specified, you will still get the `DO` loop statistics.

Note



The number of operations reported by `STATISTICS` is approximate. Each assignment, arithmetic operation, and logical operation counts as an operation. Each subscript of a subscripted variable counts as a separate operation.

To unroll the loop two times instead of four times (which is the default), use

```
$OPTIMIZE LOOP_UNROLL COPIES=2
```

To unroll a DO loop that is larger than the default, use

```
$OPTIMIZE LOOP_UNROLL COPIES=2, SIZE=500
```

substituting an appropriate size for the digit 500.

Example

C Example to illustrate the use of LOOP_UNROLL

```
$OPTIMIZE ON
```

```
PROGRAM UNROLL_EXAMPLE
```

```
DIMENSION A(10), B(10,10)
```

```
DIMENSION X(10,10,10), Y(10,10,10), Z(10,10,10)
```

```
...
```

```
...
```

C The inner loop has only one statement. The loop can be unrolled
C 10 times avoiding a branch and an extra copy of the loop. A straight
C line code is generated for the inner loop.

```
$OPTIMIZE LOOP_UNROLL COPIES=10
```

```
DO 20 J=1,10
```

```
DO 10 I=1,10
```

```
A(I) = A(I) + B(I,J)
```

```
10 CONTINUE
```

```
20 CONTINUE
```

C Change COPIES back to default.

```
$OPTIMIZE LOOP_UNROLL COPIES=4
```

```
..
```

```
..
```

C This DO loop has more than 60 operations.

C This does not get unrolled by default. The LOOP_UNROLL option is used

C to unroll it two times by increasing the SIZE to a large value.

```
$OPTIMIZE LOOP_UNROLL COPIES=2, SIZE=200
```

```
DO 40 I=1,10
```

```
DO 30 J=1,20
```

```
V1 = X(I,J+1,K) - X(I,J-1,K)
```

```
V2 = Y(I,J+1,K) - Y(I,J-1,K)
```

```
V3 = Z(I,J+1,K) - Z(I,J-1,K)
```

```
X(I,J,K) = X(I,J,K) + A11*V1 + A2*V2 +
```

```
* A3*V3 + S*(Y(I+1,J,K)-2.0*X(I,J,K)+X(I-1,J,K))
```

```

          Y(I,J,K) = Y(I,J,K) + A1*V1 + A2*V2 +
*          A3*V3 + S*(Y(I+1,J,K)-2.0*Y(I,J,K)+Y(I-1,J,K))

          Z(I,J,K) = Z(I,J,K) + A1*V1 + A2*V2 +
*          A3*V3 + S*(Z(I+1,J,K)-2.0*Z(I,J,K)+Z(I-1,J,K))
30          CONTINUE
40          CONTINUE

```

C Change the options back to the default values.

```
$OPTIMIZE LOOP_UNROLL COPIES=4, SIZE=60
```

```
. .
```

```
. .
```

```
STOP
```

```
END
```

What to Do If the Optimized Program Fails

Occasionally a program works differently after optimization. If this happens:

1. Make sure that optimizer assumptions were not violated. If they were, correct the code and recompile, or recompile the code without optimization.
2. Isolate the problem code and first try optimization with level one modules. If that does not work, recompile the code without optimization.

If the problem still occurs, contact the HP Software Support Center or your HP representative.

Programming for Portability

This chapter describes how to port FORTRAN 77 programs from other systems onto HP systems, and how to make new programs easily transportable between HP systems. The suggestions in this chapter are derived from basic concepts of structured programming and good programming practices. However, because portability is the main topic of this chapter, some of the suggestions might not result in run-time efficiency.

In general, methods for program portability fall into these five categories:

- Restricting a program to features and statements that are a part of the HP FORTRAN 77 standard.
- Making a program's data storage consistent and well-defined.
- Designing the source code so changes can be easily made to the program.
- Avoiding unstructured programming constructs and features.
- Understanding operating system issues.

Using these methods does not guarantee that your program will compile, link, and load successfully, and run on a new system exactly as it did on the previous system. Differences in the machine architecture and the operating system limit that possibility. However, following the methods in this chapter will minimize the changes you will have to make to any HP FORTRAN 77 program.

Restricting Programs to the HP FORTRAN 77 Standard

The first step in writing portable FORTRAN 77 programs is to only use features of the language that are available on every system to which you port your programs. For HP systems, this feature set is defined by the HP FORTRAN 77 standard, which fully implements the ANSI standard for FORTRAN as defined by the ANSI X3.9-1978 documents. HP FORTRAN 77 also includes all of the extensions contained in the Military Standard (MIL-STD-1753) definition of FORTRAN 77. In addition, HP has included extensions for compatibility, portability, and readability. The syntax and semantics of these extensions to the ANSI standard are described in the *HP FORTRAN 77/iX Reference*.

Do not use any feature that is not a part of the standard. When moving FORTRAN 77 programs from a non-HP system to different HP systems, identifying features that are not defined in the HP FORTRAN 77 standard can be difficult. However, you can use the ANSI compiler directive to help identify nonstandard features in HP FORTRAN 77 programs.

When the ANSI compiler directive is used in a program, all features not conforming to the ANSI standard are flagged with appropriate warning messages. An output listing with ANSI ON easily identifies all non-ANSI features. However, the ANSI directive also flags any MIL-STD-1753 or HP extensions with warnings. Therefore, it is still a tedious task to use the full capabilities of HP FORTRAN 77 if you only use the ANSI compiler directive. Most HP FORTRAN 77 compilers include additional directives that allow you to specifically define which features will be flagged. Refer to the *HP FORTRAN 77/iX Reference* for more details on the ANSI directive and for information on other compiler directives that help identify nonstandard features.

When using compiler directives to help identify nonstandard features, place the directives in the source file when your program is being developed. If this is done, any deviation from the chosen standard is immediately flagged. If you are going to transfer an existing program between systems, insert the directive and recompile the program on the new system; any nonstandard features are flagged. Also, if you modify your program in the future and recompile the program with the directives included, the nonstandard features are again flagged. Therefore, by using the ANSI directive or a system-dependent directive, it is easy to identify features that are not common to both systems.

Using Consistent Data Storage

Because machine architectures differ between systems, the methods of storing data also differ. Default sizes for data types might also differ between HP and non-HP systems, resulting in the data storage accidentally overlapping. This creates a program that compiles and loads on both HP and non-HP systems without errors, but produces different results with the same input data. However, if data storage is used carefully and consistently, programs producing varied results on different systems with the same data will be less frequent.

Guidelines for consistency and integrity of a program's data storage are described below. Using these guidelines might not produce a program that makes optimal use of space, or executes as quickly as possible; however, using these guidelines will produce code that is easily portable and is fairly efficient.

Use the LONG and SHORT Compiler Directives

If you are porting between HP systems, data type consistency is easily maintained because the implicit defaults for HP FORTRAN 77 data types on all HP implementations are the same. That is, on all systems, the type INTEGER defaults to INTEGER*4 and the type REAL defaults to REAL*4. However, if your program was developed on a non-HP compiler or on an HP compiler that is not an implementation of the HP FORTRAN 77 standard (such as FORTRAN/3000, FORTRAN 4X, or FORTRAN/1000), the default data type sizes might not be the same. For example, the type INTEGER defaults to INTEGER*2 under FORTRAN/3000.

You can use the HP FORTRAN 77 LONG and SHORT compiler directives to change the default data type sizes. The LONG directive on HP systems only documents the HP FORTRAN 77 default because the default is already four bytes for INTEGER and LOGICAL values. However, the SHORT directive sets the default for INTEGER and LOGICAL to two bytes. For example, if you insert a SHORT directive in every unit to be ported from FORTRAN/3000, the implicit data type sizes will not change and therefore should not create any problems. Refer to the *HP FORTRAN 77/iX Reference* for more information on the LONG and SHORT directive.

Consistent data storage on HP systems is obtained by using the implicit HP FORTRAN 77 defaults or by using the LONG and SHORT directives. However, because the data type sizes are not clearly documented in the program itself, these methods are not recommended for programs that might be ported more than once. Instead, declare all variables, as described later in this chapter.

Use Length Specifications in All Type Statements

Not only should all the variables be explicitly declared, but their individual sizes should be defined and documented. By declaring sizes, you avoid the possibility of different default sizes causing invalid run-time results and you ensure that the common and equivalence lengths are the same. Also, the data storage is well defined, with the exact amount of variable storage specified in the declarations. This guideline is not recommended if performance is a priority.

Declare All Variables

Implicit declaration of variables according to the standard HP FORTRAN 77 conventions (variables beginning with the letters I through N are INTEGER; the rest are REAL) causes a default data item's size to be assigned. In addition, implicit declarations are not documented, making modifications difficult. Although implicit declarations might save programming time during the initial writing phase, it might take more time for debugging and modifying the program. Therefore, declaring all variables in a program should save programming time, as well as ensuring the integrity of the data structures.

Because HP FORTRAN 77 has an implicit declaration facility defined as part of the ANSI language, you might forget to declare all variables. For example, it would be easy to forget to declare an index of a DO loop if the loop was added after the initial design was complete. You can use the IMPLICIT NONE statement to turn off the implicit declaration facility. The IMPLICIT NONE statement (a MIL-STD-1753 extension) generates error messages for undeclared variables. Placing this statement in each program unit immediately following the PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement guarantees that no variable declaration is overlooked.

Avoid Using the EQUIVALENCE Statement

Data alignment requirements differ between systems. For example, noncharacter data on some machines must be word-aligned, while character data on other machines must be aligned on a byte boundary. Considering the different machine word sizes between systems, you can see how data alignment and equivalence overlapping could change without being noticed when a program is ported between systems. Therefore, when possible, avoid equivalences between character and noncharacter data.

Because systems store data differently, an EQUIVALENCE statement can cause storage allocation problems. Complicated equivalence expressions have a higher chance of problems in the storage allocation algorithm; therefore, do not try to conserve storage by using equivalences. Also, long and complicated equivalence expressions can be confusing when changes have to be made to the program.

Declare Common Blocks the Same in Every Program Unit

Because common blocks are implemented differently on different HP systems, you should declare each common block exactly the same in every program unit in which it is used. By doing this, your program is easier to read and is consistent in data storage. If common blocks are not declared the same, some of the same problems associated with the EQUIVALENCE statement could occur. However, because the declarations are in different program units (and possibly in different files), the problems are more difficult to find and correct.

To ensure that the declarations remain identical, use the INCLUDE statement or INCLUDE compiler directive. Place each common block declaration and associated variable declarations in a separate file;

reference the file by using the INCLUDE statement or directive in every program unit that uses the common block. There is no chance of errors due to unmatched declarations if all program units that share a declaration file are recompiled whenever the declaration file changes. However, if all program units are not recompiled when their declaration file changes, differences in the old and new common block definitions might cause incorrect run-time results.

Initialize Data Before the Algorithm Begins

The initialization of memory locations varies between systems. Some systems check and initialize all allocated memory to a set value. Most systems leave the allocated memory in the same state as the previous program that used the memory. Therefore, to ensure storage portability, initialize all data before the actual algorithm begins.

Using HP FORTRAN 77, you can initialize data in two ways:

1. Use assignment statements to improve documentation and readability if your program is small, has very few variables, and does not have common blocks.
2. Use DATA statements with BLOCK DATA subprograms for initializing common blocks if you need more control over the initial values.

Refer to the *HP FORTRAN 77/iX Reference* for the syntax and semantics of these statements.

Avoid Accessing the Representation of Logical Values

The representation for the logical values .TRUE. and .FALSE. differs between implementations of HP FORTRAN 77. Therefore, avoid accessing these values (such as by using the EQUIVALENCE statement) and testing their internal values. You should also not use logical variables to pass nonlogical data (such as character data) because the clarity and readability of the program is reduced.

Maintain Parameter Type and Length Consistency

The final guideline for data storage portability is to maintain type and length consistency for subroutine and function parameters and function values being returned. For example, serious problems in data overlapping can result if a program calls a subroutine with two INTEGER*2 parameters when the subroutine expects two INTEGER*4 parameters. Because FORTRAN 77 passes all parameters by reference, the program's data area will become corrupt because the subroutine manipulates four extra bytes of data. Problems like this often appear when trying to port to another system.

Care should especially be taken for character data. The CHARACTER*(*) declaration should only be used when a routine must be called with different character parameter sizes. In these situations, the routines should be written to guarantee that the current length of a character parameter is not exceeded. The length can be passed as a separate parameter or determined in the subprogram by using the LENGTH intrinsic function. Programs should be written to avoid data corruption from inconsistent type and length of parameters. On your system, there may be a system-specific directive that helps ensure consistent parameter type and length; for details, see the *HP FORTRAN 77/iX Reference Manual*.

By following the data storage guidelines described in this section, you will avoid most data overlap and data size problems. In summary, all the variables and parameters should be declared so that any system's

storage allocation algorithm produces data areas that function identically. Alignment problems are avoided by a minimal use of the EQUIVALENCE statement. All common blocks of the same name should have the same internal structure. Data should be explicitly initialized. The internal representation of logical values should not be relied upon. Type and length consistency should be maintained for parameters and return values.

Writing Programs That Can Be Easily Modified

If a program will be moved from one system to another, it is unlikely that the program will compile, link, load, and run correctly on the first attempt. Therefore, an easily portable program should also be able to be easily changed. This section describes some guidelines for writing programs that can be easily modified.

One way to write a program that can be easily modified is to space and indent statements. Without spacing and indenting, the program is hard to read. For example, the following FORTRAN program segment does not indent or space between logical blocks:

```
J1=0
DO I=1,30
J1=J1+1
IF(A(I,J1).EQ.0)THEN
DO J=1,30
IF(J.NE.I)THEN
A(I,J)=1
ELSE
A(I,J)=0
ENDIF
END DO
ELSE IF(A(I,J1).EQ.1)THEN
DO J=1,30
IF(J.NE.I)THEN
A(I,J)=0
ELSE
A(I,J)=1
ENDIF
END DO
ELSE
A(I,J1)=999
ENDIF
END DO
```

It is not obvious what happens in the section of code, nor is it easy to see how the statements are nested. In contrast, the same program segment that indents at each nesting level and double-spaces between each logical section is shown below:

```
J1 = 0
DO I=1,30
  J1 = J1 + 1

  IF (A(I,J1) .EQ. 0) THEN
    DO J=1,30
      IF (J .NE. I) THEN
        A(I,J) = 1
      ELSE
        A(I,J) = 0
      ENDIF
    END DO

    ELSE IF (A(I,J1) .EQ. 1) THEN
      DO J=1,30
        IF (J .NE. I) THEN
          A(I,J) = 0
        ELSE
          A(I,J) = 1
        ENDIF
      END DO

    ELSE
      A(I,J1) = 999
    ENDIF

  END DO
```

The nesting levels are now obvious. The logic of the program is also easy to follow. Therefore, indenting and spacing makes a significant contribution to the readability of a program.

However, the purpose of the program segment is not obvious. One way to make a program's function clear is to use meaningful variable names. The ANSI standard restricts variable names to six uppercase letters and numbers per variable name; using this limit, meaningful names can be constructed. HP FORTRAN 77 names can be any length (with a system-specific limit on the number of significant characters) and can contain any combination of upper and lowercase letters, digits, and the underscore character. The variable must begin with a letter. The following program segment uses meaningful variable names.

```
Column_index = 0
DO Row_index=1,30
  Column_index = Column_index + 1

  IF (Matrix(Row_index,Column_index) .EQ. 0) THEN
    DO Col_count=1,30
      IF (Col_count .NE. Row_index) THEN
        Matrix(Row_index,Col_count) = 1
      ELSE
        Matrix(Row_index,Col_count) = 0
      ENDIF
    END DO

    ELSE IF (Matrix(Row_index,Column_index) .EQ. 1) THEN
      DO Col_count=1,30
        IF (Col_count .NE. Row_index) THEN
          Matrix(Row_index,Col_count) = 0
        ELSE
          Matrix(Row_index,Col_count) = 1
        ENDIF
      END DO

    ELSE
      Matrix(Row_index, Column_index) = 999
    ENDIF

  END DO
```

A glance at the segment above shows that the program performs a matrix transformation. Therefore, using meaningful symbolic names improves the understanding of a program.

Even though the readability of the example has improved by good programming practices, it is not obvious what kind of matrix transformation is taking place. To provide detailed information, use comments. A comment is identified by a C or * in column one. In addition, by appending an exclamation point (!) to the end of a source or directive line, the remainder of the line is treated as a comment. Adding comments to the example results in the following:

```

Column_index = 0

C  Check each element along the principal diagonal.

DO Row_index=1,30
  Column_index = Column_index + 1 !Avoids extra loop, retains clarity

  IF (Matrix(Row_index,Column_index) .EQ. 0) THEN

C  If the principal diagonal element is 0,
C  set all other elements in that row to 1.

    DO Col_count=1,30
      IF (Col_count .NE. Row_index) THEN
        Matrix(Row_index,Col_count) = 1
      ELSE
        Matrix(Row_index,Col_count) = 0
      ENDIF
    END DO

    ELSE IF (Matrix(Row_index,column_index) .EQ. 1) THEN

C  If the principal diagonal element is 1,
C  set all other elements in that row to 0.

    DO Col_count=1,30
      IF (Col_count .NE. Row_index) THEN
        Matrix(Row_index,Col_count) = 0
      ELSE
        Matrix(Row_index,Col_count) = 1
      ENDIF
    END DO

    ELSE
      Matrix(Row_index,Column_index) = 999 !Mark inconsistent row.
    ENDIF

  END DO

```

Now, with just a quick glance at the example, the program's function is obvious. However, be careful not to use too many comments and do not include comments that only repeat the actions of the code; these comments will obscure a program's function. Therefore, use a few, effective comments instead of many unhelpful comments.

A final method of making a program readable and easy to change is to use named constants instead of numeric or string literals. This is done by using the HP FORTRAN 77 PARAMETER statement; refer to the *HP FORTRAN 77/iX Reference* for details on the PARAMETER statement. By using named constants, the literal is documented with a meaningful name. Also, if the value needs to be changed, only the PARAMETER statement has to be changed, not every occurrence of the literal.

Adding PARAMETER statements and named constants to the example results in the following:

```
PARAMETER (Lower_bound = 1, Upper_bound = 30)
PARAMETER (Invalid_row = 999)
PARAMETER (Repl_val_a = 0, Repl_val_b = 1)
PARAMETER (Column_start = Lower_bound - 1)
.
.
.
Column_index = Column_start
```

C Check each element along the principal diagonal.

```
DO Row_index=Lower_bound,Upper_bound
  Column_index = Column_index + 1 !Avoids extra loop, retains clarity
```

```
  IF (Matrix(Row_index,Column_index) .EQ. Repl_val_a) THEN
C   If the principal diagonal element is Repl_val_a,
C   set all other elements in that row to Repl_val_b.
```

```
  DO Col_count=Lower_bound,Upper_bound
    IF (Col_count .NE. Row_index) THEN
      Matrix(Row_index,Col_count) = Repl_val_b
    ELSE
      Matrix(Row_index,Col_count) = Repl_val_a
    ENDIF
  END DO
```

```
  ELSE IF (Matrix(Row_index,Column_index) .EQ. Repl_val_b) THEN
```

```
C   If the principal diagonal element is Repl_val_b,
C   set all other elements in that row to Repl_val_a.
```

```
  DO Col_count=Lower_bound,Upper_bound
    IF (Col_count .NE. Row_index) THEN
      Matrix(Row_index,Col_count) = Repl_val_a
    ELSE
      Matrix(Row_index,Col_count) = repl_val_b
    ENDIF
  END DO
```

```
  ELSE
    Matrix(Row_index,Column_index) = Invalid_row !Mark inconsistent row
  ENDIF
```

```
END DO
```


In summary, spacing and indenting statements make the program structure clearly visible. Adding meaningful variable names give general details about a program's function. When comments are effectively used, important details become apparent. Using named constants improves documentation in the code and provides an easy way of changing values. If you apply the guidelines given above, your programs will be readable, easier to modify, and will be portable.

Avoiding Unstructured FORTRAN 77 Features

One factor in making a program transportable is to “plan for the future” in regard to changes. In general, the unstructured features of FORTRAN 77 make programs harder to understand and modify, and therefore reduce portability. Some of the unstructured features are:

- Assigned GOTO statement
- ASSIGN statement
- Computed GOTO statement
- Arithmetic IF statement
- Any use of Hollerith data
- EQUIVALENCE statement

The list above includes only the least structured features of FORTRAN 77. You should also omit any other features from your programs that you think are unstructured.

Identifying Nonstandard Features

The `STANDARD_LEVEL` compiler directive helps identify nonstandard features. This directive has three possible options: `ANSI`, `HP`, and `SYSTEM`.

The `STANDARD_LEVEL ANSI` option has the same effect as the `ANSI ON` compiler directive and flags all non-ANSI features with warning messages.

The `STANDARD_LEVEL HP` option issues warning messages for features that are not part of the FORTRAN 77 standard and gives a quick assessment of the non-HP FORTRAN 77 or system-specific features in your programs.

The default `STANDARD_LEVEL SYSTEM` option does not issue warning messages for nonstandard or system-specific features.

Refer to the *HP FORTRAN 77/iX Reference* for more details on the syntax and semantics of the `ANSI` and `STANDARD_LEVEL` compiler directives.

Avoiding Data Storage Inconsistencies

The `CHECK_FORMAL_PARM` and `CHECK_ACTUAL_PARM` compiler directives help to avoid data storage inconsistencies between a program and its subprograms. If properly used, these directives supply parameter information to the system loader or Link Editor for checking inconsistencies in type, length, and number of parameters. Refer to the *HP FORTRAN 77/iX Reference* for a complete description.

Using Comments

When your program has code that is specific to the MPE/iX operating system, place a `C` or `*` in the first column of each line of system-specific code to identify the lines as comment lines. When not compiling on MPE/iX, the code remains in the program without being executed. When the program is compiled on MPE/iX, remove the `C` or `*`.

Using Conditional Compilation Directives

Use the conditional compilation directives listed in table 8-1.

Table 7-1. Conditional Compilation Directives

Directive	Description
\$IF	Conditionally compiles blocks of code.
\$ELSE	Used with the IF directive; marks the beginning of the ELSE block of code.
\$ENDIF	Ends the IF directive.
\$SET	Assigns values to identifiers used in IF directives.

For example, the partial program below uses the system intrinsics if the program is run on an MPE/iX system and uses FORTRAN I/O for portable code.

```
PROGRAM sysmpeix

$SET (os_mpe_ix = .TRUE.)

CHARACTER buffer(80)

$IF (os_mpe_ix)
  SYSTEM INTRINSIC FREAD, FWRITE
$ENDIF
.
.
.
$IF (os_mpe_ix)
C MPE/iX intrinsic I/O; System-specific:

  length = FREAD(filenum1, buffer, 80)
  CALL FWRITE(filenum2, buffer, length, 0)

$ELSE
C FORTRAN I/O; Portable version:

  READ(5,100) buffer
  WRITE(6,200) buffer
100  FORMAT(80A1)
200  FORMAT(1X, 80A1)
$ENDIF
.
.
.
END
```

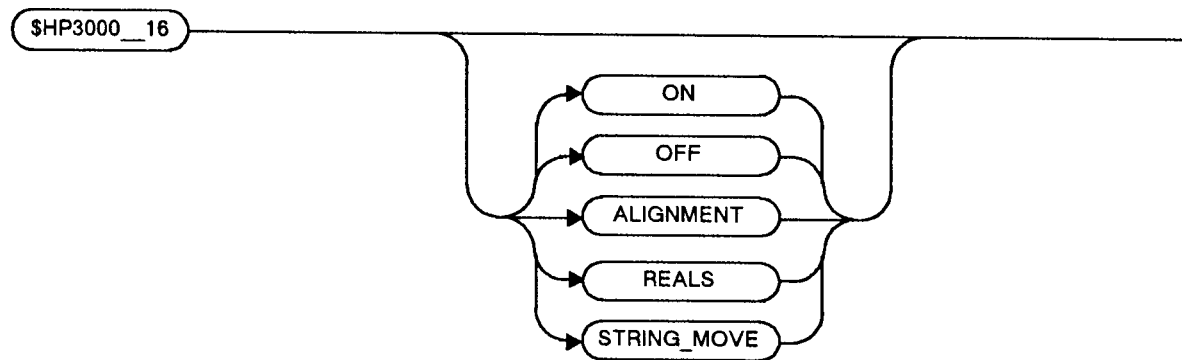
Resolving Incompatibilities between MPE V and MPE/iX: the HP3000_16

Directive

This section describes how to use the HP3000_16 compiler directive. This directive helps resolve some of the incompatibilities between the MPE V and MPE/iX operating systems and architectures. The differences between the systems are:

- Data alignment
- Floating point data
- Overlapping character substring moves

This is the syntax of the HP3000_16 compiler directive:



LG200023_016

Do not use the HP3000_16 compiler directive if the application that is ported from MPE V to MPE/iX is not affected by any of the above incompatibilities. Likewise, if the application has only one of the incompatibilities, specify only the appropriate option in the directive. Table 7-2 summarizes the options.

Table 7-2. HP3000_16 Directive Options

Option	Description
ON	Turns on all three options (ALIGNMENT, REALS, and STRING_MOVE).
OFF	Turns off all three options (ALIGNMENT, REALS, and STRING_MOVE).
ALIGNMENT	Emulates MPE V data alignment.
REALS	Emulates MPE V floating point data.
STRING_MOVE	Emulates a ripple move (which is used on MPE V) for overlapping character substrings.

Note



In this chapter, the term “HP3000 floating point” refers to Hewlett-Packard proprietary floating point data used on the 16-bit HP3000 systems. Floating point data used on the 32-bit HP3000 systems is IEEE standard.

The compiler options degrade performance, as described below. Because of the performance degradation, only use the options that are necessary.

The ON Option

This option turns on the ALIGNMENT, REALS, and STRING_MOVE options. Use the ON option only if all three options are needed.

The OFF Option

This options turns off the ALIGNMENT, REALS, and STRING_MOVE options. This is the default.

The ALIGNMENT Option

MPE V aligns noncharacter data on 16-bit boundaries and character data on 8-bit boundaries. MPE/iX aligns data on 8-, 16-, 32-, and 64-bit boundaries, depending on the data type. Table 7-3 shows the corresponding data alignments.

Table 7-3. Data Alignment on MPE V and MPE/iX

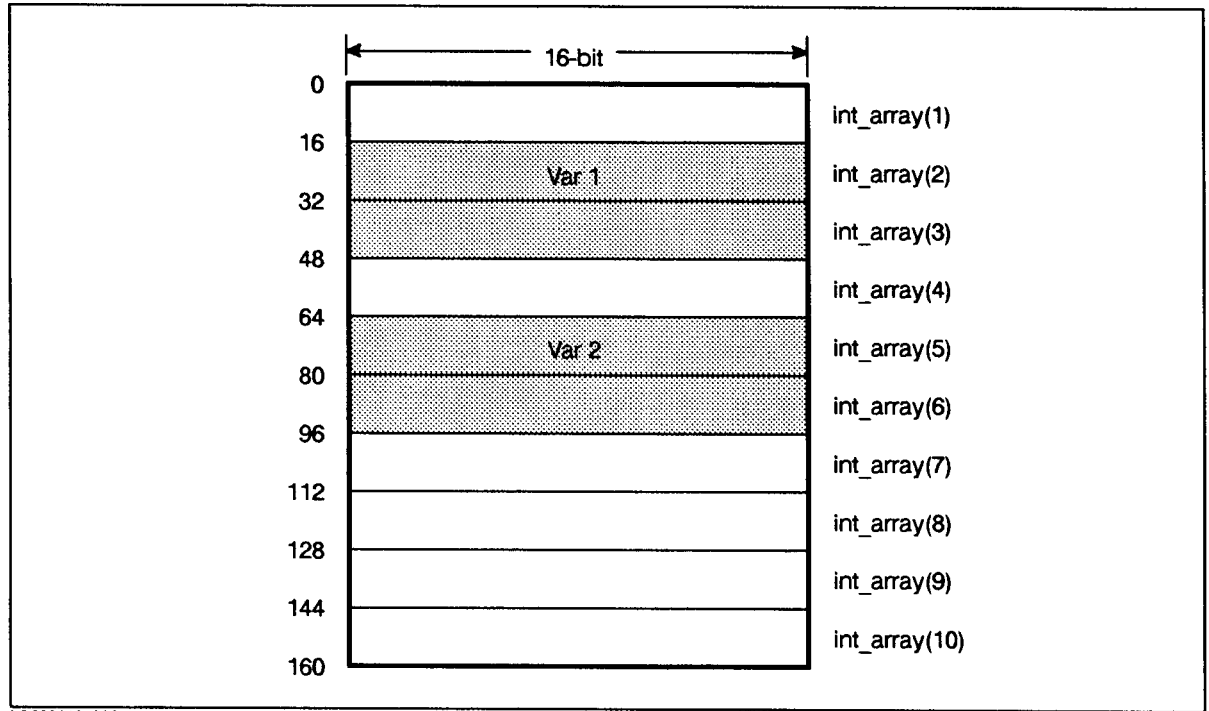
Alignment	MPE V	MPE/iX
8-bit	CHARACTER	CHARACTER, LOGICAL*1, BYTE
16-bit	COMPLEX*8, COMPLEX*16, INTEGER*2, INTEGER*4, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8	INTEGER*2, LOGICAL*2
32-bit		COMPLEX*8, INTEGER*4, LOGICAL*4, REAL*4
64-bit		COMPLEX*16, REAL*8, REAL*16

Use the ALIGNMENT option if your application assumes MPE V data packing for common and equivalence data or if the application makes calls to database intrinsics. Alternatively, the application can be modified to use MPE/iX alignment.

The following is an example of an equivalence structure that assumes 16-bit data alignment:

```
INTEGER*2  int_array(10)
INTEGER*4  var1, var2
EQUIVALENCE (var1, int_array(2)),
             (var2, int_array(5))
```

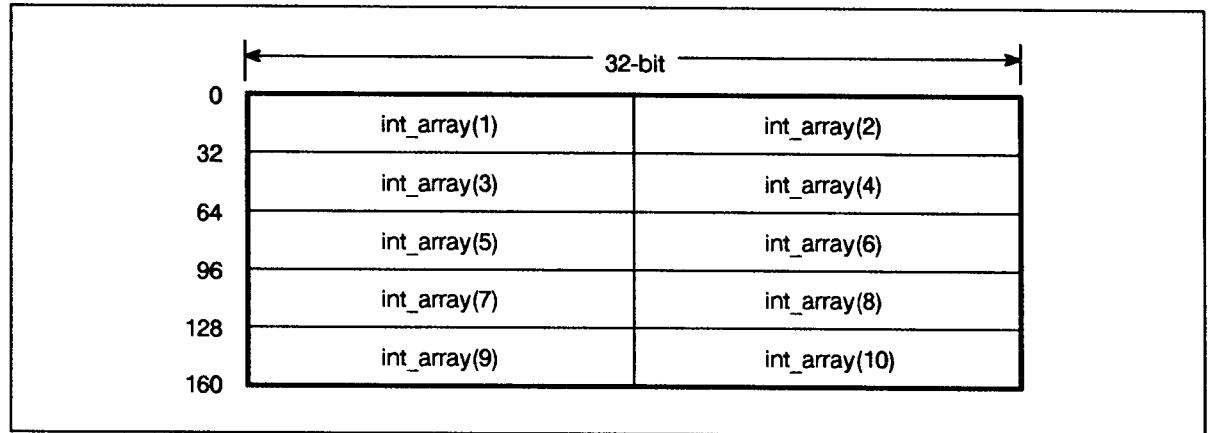
Figure 7-1 shows how the structure is stored in memory on the MPE V system.



LG200119_006

Figure 7-1. MPE V Structure

Figure 7-2 shows how the structure is stored in memory on the MPE/iX system.



LG200119_007

Figure 7-2. MPE/iX Structure

Because the INTEGER*2 (2-byte) and INTEGER*4 (4-byte) integers are stored on 16-bit boundaries on the MPE V system, the above EQUIVALENCE statement is not a problem (as shown in Figure 7-1). However, on MPE/iX, the compiler will have trouble aligning the variable `var1` on the array `int_array(2)` because `var1` should be 32-bit aligned and `int_array(2)` is on a 16-bit boundary. At this point, the compiler shifts the array `int_array` 16 bits so that it can align the variable `var1` on a 32-bit boundary. However, now the variable `var2` is not aligned on a 32-bit boundary (as shown in Figure 7-3), so the compiler issues an error message.

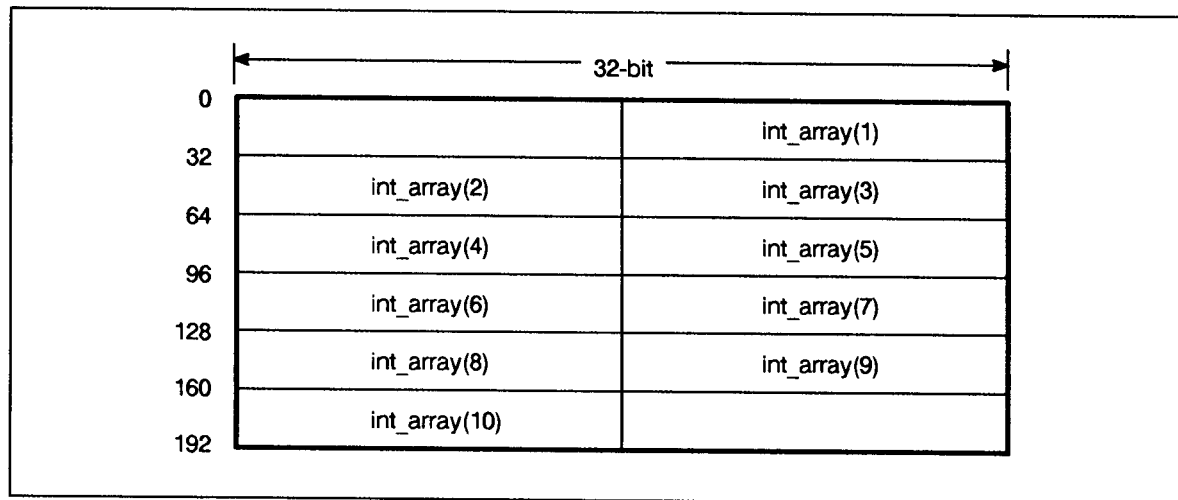


Figure 7-3. Shifted MPE/iX Structure

Using the MPE/iX default alignment yields the best performance (see Table 7-2). Using the ALIGNMENT option of the HP3000_16 directive to force 16-bit alignment of noncharacter data does not noticeably affect performance, even though three instructions for LOAD and STORE operations are needed for misaligned data versus one instruction for aligned data. Also note that all of the data in the program is not misaligned.

The REALS Option

MPE/iX uses the IEEE floating point standard for representing REAL*4, REAL*8, REAL*16, COMPLEX*8, and COMPLEX*16 types. MPE V uses proprietary HP3000 floating point to represent these types. The REALS option reads, writes, and executes all floating point numbers in the proprietary HP3000 floating point format.

Use the REALS option if your program accesses a binary flat file or accesses a database that contains proprietary HP3000 floating point. This option is not necessary if the flat files are ASCII files. When using the REALS option, all floating point data is in proprietary HP3000 floating point. Therefore, real parameters passed to external routines are in the proprietary HP3000 floating point format. If the REALS option is specified and a system intrinsic is called that requires a real parameter, a proprietary HP3000 floating point number is passed. FREAD or FWRITE can be called because the files that are being accessed contain proprietary HP3000 floating point. Some modifications might have to be made if a system intrinsic is called that expects an IEEE floating point number.

The PAUSE intrinsic and some Compiler Library, Scientific Library, V/3000, and DSG routines expect IEEE floating point real numbers. If the Compiler Library routines are declared as external or declared by the SYSTEM INTRINSIC statement, use the more efficient FORTRAN intrinsic functions instead of the compiler library routines. If you use the REALS option, the compiler calls the FORTRAN intrinsic function, which expects a proprietary HP3000 floating point number. The compiler makes the following emulation routines available:

```
em_extin'  
em_inext'  
em_hpextin (same as em_extin')  
em_hpinext (same as em_inext')  
em_pause
```

If your program calls the Compiler Library routines INEXT' or EXTIN' or calls the PAUSE system intrinsic, the easiest code change is to call the above routines. If your program calls the Scientific Library, V/3000, DSG, and IFS routines that expect IEEE floating point, the real parameters passed to these routines must be converted to IEEE floating point. Upon return, the real parameters or function return values must be converted from IEEE to the proprietary HP3000 floating point. The FPCONVERT routine converts floating point numbers in either direction.

When you use the REALS option of the HP3000_16 directive, the floating point emulation routines must be used on proprietary HP3000 floating point numbers. There can be major performance degradation if your program uses a lot of floating point arithmetic.

The **STRING_MOVE** **Option**

The `STRING_MOVE` option should only be used when the application assumes that overlapping character substring moves have a ripple effect, as on MPE V. MPE/iX does not ripple the overlapping character substring moves and therefore increases performance on character moves. For example, the result of the program fragment

```
character ch*10
ch(1:1) = '*'
ch(2:10) = ch(1:9)
```

depends on the operating system. On MPE V, the character string `ch` is filled with asterisks (*). On MPE/iX, the first and second positions contain asterisks and the remainder of the string is undefined.

Do not use this option if your program does not rely on the ripple effect of character substring moves. If there are overlapping character substrings when you use the `STRING_MOVE` option, the string is moved one byte at a time. Without this option, a fast move is used. For example, in a fast move of 20 characters, two sets of eight characters are moved, followed by one set of four characters.

Interfacing with Other Languages

This chapter describes how to call routines written in HP Pascal/iX, HP COBOL/iX, or HP C/iX from an HP FORTRAN 77/iX program, and how to call an HP FORTRAN 77/iX subroutine or function from HP Pascal/iX, HP COBOL/iX, or HP C/iX.

If you call other languages from HP FORTRAN 77/iX, the actual parameters of the internal procedure or function must match the formal parameters of the external procedure or function. The CHECK_ACTUAL_PARM compiler option determines the level of the checking information placed in the object file for the Link Editor when performing a LINK or ADDXL. Refer to the *HP FORTRAN 77/iX Reference Manual* for more information on the CHECK_ACTUAL_PARM compiler directive.

When calling HP FORTRAN 77/iX from other languages, you must match the parameters appearing in the non-FORTRAN 77 program with the formal parameters of the external FORTRAN 77 procedure or function. The CHECK_FORMAL_PARM compiler option determines the checking information placed in the object file for the Link Editor when performing a LINK or ADDXL. Refer to the *HP FORTRAN 77/iX Reference Manual* for more information on the CHECK_FORMAL_PARM compiler option.

All parameters passed to an HP FORTRAN 77/iX subprogram must be passed by reference. By default, HP FORTRAN 77/iX passes parameters of all types by reference, except for character data. When passing character data, a descriptor is passed that includes the address and length of the string. HP FORTRAN 77/iX allows parameters to be passed by value to invoke non-FORTRAN 77 program units that allow passing arguments by value. When passing parameters by value, the ALIAS compiler directive must indicate how each parameter is passed.

HP FORTRAN 77/iX does not allow arrays to be passed by value. Therefore, an HP FORTRAN 77/iX program cannot call a non-FORTRAN 77 program unit that requires an array parameter to be passed by value. Also, when you call functions that have no parameters, you must specify an empty parameter list, ().

HP Pascal/iX

HP Pascal/iX is the ANSI standard version of Pascal for the HP 3000 Series 900 computer.

An HP Pascal/iX procedure or function can be called from an HP FORTRAN 77/iX program and an HP FORTRAN 77/iX program can call an HP Pascal/iX procedure or function if the data types of the parameters match (see Table 8-1). The language code of the ALIAS compiler directive should be used for correctly passing parameters.

Table 8-1. HP FORTRAN 77/iX and HP Pascal/iX Types

HP FORTRAN 77/iX Type	HP Pascal/iX Type
INTEGER*2	SHORTINT Integer subrange in the range -32768 to +32767
INTEGER*4	INTEGER Integer subrange in the range -2147483648 to +2147483647
REAL*4	REAL
REAL*8	LONGREAL
CHARACTER*1	CHAR
CHARACTER*n	PACKED ARRAY [1..n] OF CHAR
LOGICAL*1 (BYTE)	Integer subrange in the range -128 to +127
LOGICAL*2	Integer subrange in the range -32768 to +32767 SET (1 word)
LOGICAL*4	Integer subrange in the range -2147483648 to +2147483647 SET (2 words)
COMPLEX*8	RECORD real_part : REAL; imag_part : REAL; END;
COMPLEX*16	RECORD real_part : LONGREAL; imag_part : LONGREAL; END;

Calling HP Pascal/iX from HP FORTRAN 77/iX

HP FORTRAN 77/iX cannot pass arrays by value, so you cannot call an HP Pascal/iX routine with a value parameter of a type corresponding to an HP FORTRAN 77/iX array type. You must use the %VAL parameter of the ALIAS compiler directive for other types of HP Pascal/iX value parameters.

All data transferred between HP FORTRAN 77/iX and HP Pascal/iX must be passed through parameter lists because HP FORTRAN 77/iX cannot specify global variables and HP Pascal/iX cannot specify common blocks. The calling HP FORTRAN 77/iX program can have a common area, but the external HP Pascal/iX procedure or function cannot access this common area.

Parameter checking should be turned off because HP Pascal/iX generates different type check values from HP FORTRAN 77/iX values. To turn off the checking, specify \$CHECK_ACTUAL_PARM 0\$ in the HP FORTRAN 77/iX program, or specify \$CHECK_FORMAL_PARM 0\$ in the HP Pascal/iX procedure.

HP FORTRAN 77/iX program that calls an HP Pascal/iX procedure:

```
PROGRAM call_pascal

c Calling an HP Pascal/iX procedure

$ALIAS pasprog PASCAL(%REF)
CHARACTER str*30
str='Pass this string'
CALL pasprog(str)
PRINT *,str
END
```

External HP Pascal/iX procedure:

```
$SUBPROGRAM$
PROGRAM pascal;
TYPE charstr = PACKED ARRAY[1..30] OF CHAR;

{ Turn parameter checking off because HP Pascal/iX generates different
  parameter type check values than HP FORTRAN 77/iX.          }

$CHECK_FORMAL_PARM 0$
PROCEDURE pasprog(VAR str:charstr);
VAR output : TEXT;
BEGIN

{ Open OUTPUT so we can display the string to verify that
  it was passed correctly                                     }

REWRITE(output, '$STDLIST');
WRITELN(output, str);

{ Add to the string }
strmove(strlen(' back again'), ' back again', 1, str, 17);
END;
BEGIN
END.
```


Calling HP FORTRAN 77/iX from HP Pascal/iX

An HP FORTRAN 77/iX subroutine or function can be called from an HP Pascal/iX program if the data types of the parameters match (see Table 8-1). However, be careful when passing character strings. HP FORTRAN 77/iX expects an additional word that describes the maximum length of the string, while PAC's (packed array of *char*) in Pascal do not. When an HP Pascal/iX character string is passed to HP FORTRAN 77/iX, the compiler expects the string to be passed by reference (the address of the string) and then expects the maximum length of the string to be passed by value. If the HP FORTRAN 77/iX routine is declared EXTERNAL FTN77 in the HP Pascal/iX program, the length is automatically passed as HP FORTRAN 77/iX expects it.

HP Pascal/iX cannot access an HP FORTRAN 77/iX common area and cannot pass a file or a label to an external HP FORTRAN 77/iX routine.

The following example shows how to pass character strings between HP Pascal/iX and HP FORTRAN 77/iX.

HP Pascal/iX program that calls an HP FORTRAN 77/iX subroutine:

```
PROGRAM callfort(OUTPUT);
CONST str_stuff='Pass this string to FORTRAN 77';
TYPE pac = PACKED ARRAY[1..50] OF CHAR;
VAR str : pac;
    cur_len:integer;
{ Declare the external HP FORTRAN 77/iX program as EXTERNAL FTN77 so the
  PAC is passed correctly and so compatible data type information is
  generated for the Link Editor. Two parameters are passed: the
  the PAC by reference (or the address of the string) and a one-
  word integer by reference, which is the current length of the PAC.
  HP Pascal/iX passes the maximum length of str (50 in this example)
  by value between these two arguments to satisfy HP FORTRAN 77/iX
  requirements for passing character data. }

PROCEDURE fortprog(VAR str:pac;
                  VAR cur_len:integer);
EXTERNAL FTN77;
BEGIN
str:=str_stuff;
{ Get the current length of the PAC }
cur_len:=strlen(str_stuff);
WRITELN(str);

{ Call the HP FORTRAN 77/iX subroutine and pass the PAC
  and the current length of the PAC }

fortprog(str,cur_len);

{ Do a linefeed to print the concatenated string on the following line}
```

```
WRITELN;  
WRITELN(str);  
END.
```

HP FORTRAN 77/iX subroutine:

```
      SUBROUTINE fortprog(str,cur_len)
c The formal parameters are the character string and the current
c length of the string; the maximum length of the character
c string is a hidden parameter that HP FORTRAN 77/iX uses.
      IMPLICIT NONE
      INTEGER*4 cur_len
c Use maximum length (the 2nd actual parameter) as the character length:
      CHARACTER str*(*)
c Concatenate the strings and print result
      str = str(1:cur_len) // ' and then back again'
      PRINT *,str
      RETURN
      END
```

HP COBOL II/iX

The data types of HP FORTRAN 77/iX and HP COBOL II/iX differ. Numeric HP COBOL II/iX data types are binary packed-decimal or in ASCII format (see Table 8-2). However, by taking the size and the format into consideration, you can successfully match HP FORTRAN 77/iX and HP COBOL II/iX types.

Table 8-2. HP COBOL II/iX Numeric Types and Formats

HP COBOL II/iX Type	Description of the Format
COMP-3	Packed decimal format with the sign in the rightmost half-byte and 2 digits per byte.
COMP	Binary format; the sign bit 0 is for positive, 1 for negative. The size S9 to S9(4) is 2 bytes. The size S9(5) to S9(9) is 4 bytes. The size S9(10) to S9(18) is 8 bytes.
DISPLAY	Unpacked decimal format (ASCII). Unsigned: alphanumeric format; no leading or trailing sign; 1 character per byte. Sign is leading: alphanumeric format; sign overpunched in leftmost byte. Sign is trailing: alphanumeric format; sign overpunched in rightmost byte. Sign is leading and is separate: first byte is ASCII '-' for negative and '+' for positive. Sign is trailing and is separate: last byte is ASCII '-' for negative and '+' for positive.

Table 8-3 shows examples of possible matches between HP COBOL II/iX and HP FORTRAN 77/iX types.

Table 8-3. HP COBOL II/iX and HP FORTRAN 77/iX Data Types

HP COBOL II/iX Type	HP FORTRAN 77/iX Type
PIC X(N)	CHARACTER*n
PIC S9(01)-S9(04) COMP	INTEGER*2 {-9999..9999}
PIC S9(05)-S9(09) COMP	INTEGER*4 {999,999,999..999,999,999}
PIC S9(10)-S9(18) COMP	INTEGER*4 varname(2)

The HP COBOL II/iX types 01 and 77 always start on word boundaries.

Calling HP COBOL II/iX from HP FORTRAN 77/iX

HP COBOL II/iX expects parameters to be passed by reference. When passing character data to an HP COBOL II/iX routine, the ALIAS compiler directive must indicate that the routine language is HP COBOL II/iX, so only the string address is passed and not the additional length. For example,

HP FORTRAN 77/iX program that calls an HP COBOL II/iX subprogram:

```
$ALIAS cobsubr COBOL

      PROGRAM fortran_cobol
      IMPLICIT NONE
      INTEGER*4 int1,int2,int3
```

C By default, all parameters are passed by reference.

```
      int1 = 25000
      int2 = 30000
      CALL cobprog(int1,int2,int3)
      PRINT *,int3
      END
```

HP COBOL II/iX subprogram:

```
$CONTROL SUBPROGRAM
IDENTIFICATION DIVISION.
PROGRAM-ID. COBPROG.
AUTHOR. LD.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000.
OBJECT-COMPUTER. HP3000.
DATA DIVISION.
LINKAGE SECTION.
77 IN1      PIC S9(09) COMP.
77 IN2      PIC S9(09) COMP.
77 OUT      PIC S9(09) COMP.
PROCEDURE DIVISION USING IN1, IN2, OUT.
PARA-1.
      ADD IN1, IN2, GIVING OUT.
      GOBACK.
```

**Calling HP FORTRAN
77/iX from HP COBOL
II/iX**

The GIVING phrase must be used when calling an HP FORTRAN 77/iX function from HP COBOL II/iX.

Example 1

HP COBOL II/iX program that calls an HP FORTRAN 77/iX function:

```
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 subroutine "LARGER" to find the largest element *
027000* 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 subroutine "BACKWARDS" to reverse a string of *
036000* 10 characters. *
037000* Shows passing character strings to FORTRAN subroutine *
038000*****
039000
040000     MOVE 1 TO LEN.
041000     DISPLAY STRING-1 " BACKWARDS IS " WITH NO ADVANCING
042000     CALL "BACKWRDS" USING STRING-1 \LEN\.
043000     DISPLAY STRING-1.

```


HP FORTRAN 77/iX function:

```
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 .GT. A(I)) GO TO 100
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
```

Example 2

HP COBOL II/iX program that calls an HP FORTRAN 77/iX function:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.    CALLFTN2.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01  INT-1      PIC S9(4) COMP SYNC VALUE 13.  
01  INT-2      PIC S9(4) COMP SYNC.  
01  STRING-1   PIC X(10) VALUE "0123456789".  
PROCEDURE DIVISION.  
P1.  
    CALL "FUNC1" USING INT-1,@STRING-1 GIVING INT-2.  
    DISPLAY STRING-1.  
    DISPLAY INT-2.
```

HP FORTRAN 77/iX function:

```
$FTN3000_66 CHARS ON  
    INTEGER*2 FUNCTION func1(i,string)  
    CHARACTER string*10  
    INTEGER*2 i  
    WRITE(6,*) i, string  
    string = 'This is it'  
    func1 = i  
    END
```

HP C/iX

HP C/iX, when invoked in ANSI mode, is a conforming implementation of ANSI C, as specified by American National Standard X3.159-1989. It runs on the HP 3000 Series 900 computer.

An HP C/iX procedure or function can be called from an HP FORTRAN 77/iX program and an HP FORTRAN 77/iX program can call an HP C/iX procedure or function if the data types of the parameters match (see the table below). The ALIAS compiler directive should be used for correctly passing parameters.

Table 8-4. HP FORTRAN 77/iX and HP C/iX Types

HP FORTRAN 77/iX Type	HP C/iX Type
—	char
CHARACTER*1	unsigned char
INTEGER*2	short
—	unsigned short
INTEGER*4 or INTEGER	int
—	unsigned int
INTEGER*4	long
—	unsigned long
REAL or REAL*4	float
REAL*8 or DOUBLE PRECISION	long float
REAL*8 or DOUBLE PRECISION	double
COMPLEX or COMPLEX*8	<i>(See Note 1)</i>
DOUBLE COMPLEX or COMPLEX*16	<i>(See Note 2)</i>

HP FORTRAN 77/iX and HP C/iX Types (Continued)

HP FORTRAN 77/iX Type	HP C/iX Type
INTEGER*4	enum
Not available.	pointer to <i>type</i> long pointer to <i>type</i>
CHARACTER*n (See Note 3)	string (char *)
CHARACTER*1 array (See Notes 4 & 5)	char array
Hollerith array (See Note 4)	<i>See Note 5)</i> arrays
LOGICAL*2 (See Note 6)	short (Used for logical test)
LOGICAL*4 (See Note 6)	int (Used for logical test)
Used when calling a SUBROUTINE	void
Used when calling a FUNCTION	function

Notes on HP FORTRAN 77/iX and HP C/iX Types

1. The FORTRAN 77 type of COMPLEX or COMPLEX*8 is equivalent to the following HP C/iX structure:

```
struct complex {
    float real_part;
    float imag_part;
};
```

2. The FORTRAN 77 type of DOUBLE COMPLEX or COMPLEX*16 is equivalent to the following HP C/iX structure:

```
struct complex {
    double real_part;
    double imag_part;
};
```

3. HP FORTRAN 77 passes character strings as parameters using string descriptors corresponding to the following HP C/iX declarations:

```
char *char_string; /* points to string */
int len;           /* length of string */
```

4. HP FORTRAN 77/iX stores arrays in column-major order whereas HP C/iX stores arrays in row-major order. The default lower bound for HP FORTRAN 77 is one; for HP C, the lower bound is always zero.
5. HP FORTRAN 77 does not terminate character or Hollerith strings with a null byte, but HP C does.
6. HP FORTRAN 77 and HP C do not share a common definition of *true* or *false*. In HP FORTRAN 77, logical values are determined by the low-order bit of the high-order byte. If this bit is 1, the logical value is *.TRUE.*, and if the bit is zero, the logical value is *.FALSE.* HP C uses any nonzero value to represent *true* and uses zero for *false*.

Files and I/O

A FORTRAN unit cannot be passed to a C routine to perform I/O on the associated file. Nor can a C file pointer be used by a FORTRAN routine. However, a file created by a program written in either language can be used by a program of the other language if the file is declared and opened within the latter program.

Mixing FORTRAN direct, terminal, or tape READ statements with `stdio fread` input results in the FORTRAN READ commencing from the beginning of the next block after the contents of the buffer, not from the current position of the input cursor in the `fread` buffer. The same situation in reverse may occur by mixing `read` with a FORTRAN sequential disk read.

Parameter Passing between HP FORTRAN 77 and HP C

The major difference is that FORTRAN and C pass parameters differently—FORTRAN by *reference* and C by *value*. This means that all actual parameters in an HP C call to an HP FORTRAN 77 routine must be pointers or variables prefixed with the unary address-of operator `&`. In addition, all formal parameters in an HP C routine called from HP FORTRAN 77 must be pointers, unless you use the `$ALIAS` directive in the HP FORTRAN 77 code to change FORTRAN's parameter passing mechanism so the parameters are defined as value parameters. Refer to the *HP FORTRAN 77/iX Reference*, chapter 7 for more information about the `$ALIAS` directive.

To pass string variables of any length from an HP C call to an HP FORTRAN 77 subroutine you must build and pass a two-parameter descriptor (defined in Note 3 above), initialize the string appropriately, and pass two arguments. The two arguments are the *pointer* to the characters and the value of the length word. This is shown below:

```
/* C program */
extern print_str();
main()
{
    char *str = "ABCDEFGH";
    int len;

    len = strlen (str);
    (void) print_str (str, len);
}
```

```
C    FORTRAN program
C
C    SUBROUTINE print_str (str, len)
C    ASSUME MAX LENGTH OF 300
C    character*300 str
C    integer len

    if (len .GE. 5) then
        print *, str(1:5)
    else
        print *, str(1:len)
    endif
END
```

This example shows passing a character string from a FORTRAN program to a C function. The function returns the number of characters in the string before a space. Otherwise it returns the maximum string length.

```
/* C program */
#define MSLEN 300

sizer(x) char *x;
{
    register int i;

    for (i=0; i <MSLEN; i++)
        if (x[i] == ' ') return(i);
    return(MSLEN);
}
```

```
C FORTRAN program
$alias sizer='sizer'(%ref)
program test
character*300 x
integer sizer
external sizer
integer i
data x/"abcdefghi klmnop"/

i = sizer(x)
print *,i
end
```

Using System Intrinsic

MPE/iX system intrinsics are procedures and functions written in HP Pascal/iX. The system intrinsics handle individual programming operations and are invoked by procedure or function calls. This chapter describes how to define and call the system intrinsics.

FORTTRAN 77 intrinsic functions are part of the FORTRAN 77 library in accordance with the ANSI and MIL-STD-1753 standards. The FORTRAN 77 intrinsics do not have to be defined as system intrinsics. Do not confuse MPE/iX system intrinsics with FORTRAN 77 intrinsics; this chapter only discusses system intrinsics. Refer to the *HP FORTRAN 77/iX Reference Manual* for a description of the FORTRAN 77 intrinsics. Refer to the *MPE/iX Intrinsics Reference Manual* and the *HP FORTRAN 77/iX Reference Manual* for a full description of the system intrinsics.

In addition to the system intrinsics defined in the *MPE/iX Intrinsics Reference Manual*, each subsystem (such as VPLUS, HPIMAGE, etc.) can define procedures or functions in the SYSINTR file. You can also create intrinsic declarations for your own procedures or functions via the HP Pascal/iX compiler using the BUILDINT directive.

Defining System Intrinsic

To define a system intrinsic in a FORTRAN 77 program, place the SYSTEM INTRINSIC statement before any executable statement in each procedure or function in which the intrinsic is used. Alternatively, the SYSTEM INTRINSIC directive can be used with the same result as the SYSTEM INTRINSIC statement, except that the directive has a global effect. The SYSTEM INTRINSIC directive must appear before the first nondirective statement of the program unit in which it is to start taking effect. It is not necessary to define how the parameters are passed because FORTRAN 77 gets this information from the SYSINTR file for system intrinsics. For example,

```
C SYSTEM INTRINSIC directive:

$SYSTEM INTRINSIC calendar      ! Only needs to be declared once;
                                ! does not have to be redefined if
                                ! subsequent functions of procedures
                                ! call calendar.

C SYSTEM INTRINSIC statement:

    SYSTEM INTRINSIC calendar    ! Must be declared in each
                                ! procedure or function
                                ! that calendar is used.
```


Matching Actual and Formal Parameters

When a procedure or function is identified as an intrinsic, the formal parameters do not have to be listed. When an intrinsic is called, the compiler checks the SYSINTR file to compare the actual parameters with the formal parameters. Table 8-5 shows how HP FORTRAN 77/iX actual parameters are matched to the intrinsic formal parameters.

Table 8-5. HP FORTRAN 77/iX and HP Pascal/iX Data Types

HP FORTRAN 77/iX Data Type	Corresponding HP Pascal/iX Data Type	Description
INTEGER*2	SHORTINT or range type (User-defined)	16-bit signed integer (I16)
INTEGER*4	INTEGER or range type (User-defined)	32-bit signed integer (I32)
Not available	LONGINT or range type (User-defined)	64-bit signed integer (I64)
INTEGER*2	BOOLEAN or range type (User-defined)	16-bit unsigned integer (U16)
INTEGER*4	User-defined range type	32-bit unsigned integer (U32)
Not available	User-defined range type	64-bit unsigned integer (U64)
REAL*4	REAL	32-bit real (R32)
REAL*8	LONGREAL	64-bit real (R64)
LOGICAL*2	BOOLEAN	Boolean (B)
CHARACTER	CHAR	Character (C)
INTEGER*4	LOCALANYPTR	32-bit address (@32)
Not available	GLOBALANYPTR	64-bit address (@64)
Equivalent array type	ARRAY (any type)	Array (A)
Array type	RECORD (any type)	Record (REC)

Example

```
PROGRAM intrinsic_example

IMPLICIT NONE

SYSTEM INTRINSIC fopen,read,fgetinfo,fclose,printfinfo
SYSTEM INTRINSIC fcontrol

CHARACTER*36 filename,cmd,fname
CHARACTER tab*10
INTEGER*2 ifnum,recsize
INTEGER*2 tlen,i
INTEGER*4 eof

1  FORMAT (1X)
2  FORMAT (A,'Input file > ')
3  FORMAT (1X,'The file ',A8,' has record length of ',I3)
4  FORMAT (1X,'and contains ',I3,' records.')

PRINT *, ' '
tab = ' '
eof = 0
recsize = 0
WRITE(6,1)
WRITE(6,2) tab
tlen = read(cmd,-36)

filename = cmd(1:tlen) // ' '
PRINT *, 'Input = ', filename

C  Open the old permanent file with exclusive update access

ifnum = fopen(filename,1B,105B)

IF (CCODE()) 5,6,5
5  STOP 'Open failed'
6  CALL fcontrol(ifnum,5,0B) ! Call fcontrol to rewind the file

IF (CCODE()) 7,8,7
7  CALL printfinfo(ifnum)
8  CALL FGETINFO(ifnum,,,,,recsize,,,,,eof) ! Get file information

IF (CCODE()) 10,9,10
9  WRITE(6,3) filename,recsize
WRITE(6,4) eof
10 CALL FCLOSE(ifnum,1,0)

STOP
END
```

Some MPE/iX intrinsics are OPTION EXTENSIBLE, which means that a partial formal parameter list can be passed to the intrinsic. The example above passed a partial parameter list to FOPEN and FGETINFO. The MPE/iX intrinsic FGETINFO is an option extensible intrinsic with up to 20 parameters. In the example above, the statement

```
CALL FGETINFO(ifnum,,,recsize,,,,,eof)
```

does not use the second through the fourth or the sixth through the tenth parameters. Commas must be in the parameter list to inform the compiler that the first, fifth, and eleventh parameters are the only parameters being passed. Also note that you do not have to list any parameters following the eleventh parameter if they are not needed.

Debugging FORTRAN 77 Programs

HP FORTRAN 77/iX programs can be symbolically debugged using one of the following:

- xdb symbolic debugger
- HP Toolset/iX

This chapter describes both methods of debugging.

Using xdb

The symbolic debugger `xdb` is a powerful debugging facility. Refer to the *HP Symbolic Debugger/iX User's Guide* for a complete description of `xdb`.

There are two ways to compile your program with symbolic debugger information:

- Use the *info-string* option to specify the symbolic debugger option when you compile your program. For example:

```
:FTNXLLK test_xdb,testprog;INFO="symdebug xdb"
```

- Embed the symbolic debugger option in the first statement of your source code. For example:

```
$SYMDEBUG XDB ON
PROGRAM test_debug
.
.
declarations
statements
.
.
END
```

To start the debugger on a program called `test_xdb`, execute the following command:

```
xdb test_xdb
```

The symbolic debugger (**xdb**) is the primary tool for debugging a program that does not execute properly. The debugger supports debugging capabilities on C, FORTRAN, and Pascal programs.

In addition to the symbolic debugger, HP FORTRAN 77 offers a range checking option for detecting run-time errors, as described earlier in this chapter.

To analyze a program, the debugger uses the executable file and related source files.

The debugger has many commands for viewing and manipulating your program. This section discusses how you can use it to:

- Look at the execution stack
- Look at the contents of your source files
- Look at data values
- Control execution of your program with both single step execution and the use of breakpoints

This section presents some basic *getting started* information for using the symbolic debugger.

Table 9-1 lists some simple **xdb** commands that are described in this chapter.

Table 9-1. Sample xdb Commands

Command	Description
r	Run the program
b 82	Set a breakpoint at line 82
c	Continue running until the next breakpoint
s	Single step through the next source line
t	Print a trace of the current execution stack
v	View a “window” of lines
/string	Search forward in the source for string
p abc	Print the value of variable abc
p abc = 2.2	Assign a new value to abc
p buffer\10d	Print the first 10 elements of array buffer in decimal format
q	Quit the debugger

The Strategy

When your program does not execute properly, use the following strategy to locate and correct any problems:

1. Invoke the symbolic debugger. The result is that you create a debugging environment in which you can now execute your program.
2. Execute the program in the debugging environment and then use the debugger commands to locate the execution bugs.
3. Once you have located the problem code, exit the debugger, retrieve the appropriate source files, make the required changes, recompile, and relink the program.
4. If your program still executes incorrectly, go back to running the debugger.

Program Requirement

Before the symbolic debugger can be used to analyze a FORTRAN program, you must compile the program with the SYMDEBUG XDB directive.

This ensures that necessary debugging information is incorporated into the object code. Do not use the optimizer while debugging code because the compiler cannot generate debug information and perform optimizations at the same time.

Linker and Debugger Interaction

You should be aware that using the symbolic debugger greatly increases the size of your program (often by a factor of two or three) because of the tables it creates. When you execute your program in the debugger environment, three tables are generated:

- A debug name and type table
- A value table
- A source line table

The debugger uses the first two tables to store information about program status and data values, and it uses the latter to associate lines of source code with object code.

Code for updating these tables is incorporated into a program's object code. The result is that, once created by the debugger, the tables are updated every time the linker is invoked to link the object files. The linker does not allow you to inhibit this updating.

When your program is executing correctly and you no longer need the symbolic debugger, you must recompile the source files and then relink the object files to remove the debugging information. Without the maintenance of the three debugger tables, the program occupies less space and links faster. Also, in a program's production version, you probably do not want to supply the debugger capabilities provided by the three tables.

In some cases it may be necessary to maintain two versions of a program: one with debugging information for software support purposes and a production version that has the debugging information removed. If you want the program to be *sharable*, it is necessary to remove the debugging information from the production version because the debugger does not work on sharable code.

Invoking the Debugger

Once you have prepared your program for the debugger and it is in an executable file (in the following example called ALL), invoke the `xdb` debugger by typing

```
xdb ALL
```

Note that `xdb` returns the number of procedures and, if you are not using an HP terminal, prints the first executable line. If you are using an HP terminal, the screen displays the first executable line of code (surrounded by text) centered in the source window, a line with the file, procedure, and line number information, and a command window. Then `xdb` waits for you to input a command like

```
s
```

(which allows you to step through one line of code at a time).

Exiting the Debugger

To exit the debugger, type

```
q
```

You are prompted for confirmation that you really want to exit from `xdb`. This is a safeguard in case you accidentally type the letter `q`.

Executing Your Program

To execute your program in the debugger environment, type

```
r
```

The `r` command allows you to specify any arguments that your program needs. Simply follow the command with the argument list:

```
r; info="info string"; parm=375
```

Redirection arguments can be supplied as follows:

```
r; STDIN=READDATA; STDOUT=OUTFILE
```

This means that the input is coming from the file `readdata` and the output is to be placed in the file `outfile`.

The program now has control and `xdb` waits for the program to terminate or to receive a signal. A signal such as an interrupt from the terminal, a memory fault, or a breakpoint in the program causes control to return to `xdb` and the program to suspend.

If you need to kill the program process, type

```
k
```

which is interpreted by `xdb` as an interrupt signal.

Viewing the Execution Stack

The debugger supports an environment that provides detailed information about the execution of your program. If the program halts incorrectly, the debugger's execution stack can be useful in determining the reason. Each time a routine is executed, the routine name is placed on the stack. The bottom of the stack is always the main program routine and the top is the name of the routine in which the program halted. To view the debugger stack, type

```
t
```

The `t` command causes a maximum of 20 routine calls to be printed, starting from the top of the stack. You can change the number of routine calls shown by typing

```
t n
```

where *n* is the number of called routine names you want to see (a stack depth of *n*).

The `T` command works the same as the `t` command, but the `T` command also prints the values of the local variables for each of the routines on the stack.

Viewing the Source File

The `xdb` debugger has several commands that you can use to look at your program source file while you are in the debugger environment. The debugger keeps track of the last file, routine, and line viewed (referred to as the *current file*, *current routine*, and *current line*). The debugger uses the last location viewed to determine the effect of several of its commands.

The last viewed current values are not the same as the location of the next line to be executed in the program. Changing the current file, current routine, or current line by moving around in a source file does not change the pointer to the next execution line. However, when program execution is suspended, the last line executed becomes the current line and the file and routine that contain it become the current file and current routine.

To find out what the current file, routine, and line number values are, type

```
v
```

The values are displayed in the form

```
file:procedure:line_number
```

To view a different file or routine, use the file or routine name as an argument with the `v` command. For example, the command

```
v filename
```

makes *filename* the current file and displays its first executable line. Similarly,

```
v procedurename
```

makes *procedurename* the current routine.

Once you are in the file or routine you want to view (that is, once it is the current file), you can move around in the file with the `p`, `+`, `-`, `/`, and `?` commands. All of these commands change the current line value and are described in the following sections.

The View Command

The form of the view (`v`) command is

```
v line
```

where *line* is the line number of the first line you want printed in the current file. If you do not specify *line*, the default is the current line. Thus, typing `v` causes the current line to be printed in the center of the source window, surrounded by the text file.

After using the `v` command, the new current line becomes the line following the last line printed. If the line just printed is the last line in the file, that line is the current line.

The Window Command

The window (**w**) command changes the size of the source window to a new value of n ; n can range from 1 to 20. Changing the size of the source window also changes the size of the command window.

The form of the command is

```
w n
```

The default value for n is two-thirds the length of the screen, minus one. For most HP terminals, n is 15.

The Move Command

The commands for moving around in a file, relative to the current line, are **+** and **-**. The **+n** command moves n lines past the current line and the **-n** command moves n lines before the current line. Specify the number of lines you want to move by immediately following the command with the number. If you do not specify the number of lines to move, the number defaults to one.

For example,

```
+9
```

moves nine lines after the current line and the line you move to becomes the new current line.

The Search Commands

The **/** command searches forward through the current file and the **?** command searches backward. Follow both commands with the string you want to search for. For example,

```
/doggie
```

causes a forward search for the string **doggie**.

Wild card characters and regular expressions that are supported by some text editors are not supported by the debugger. You *must* literally specify the string. If you do not specify a search string, the string previously specified is used. Searches wrap around the beginning and end of the file. When a search string is located, the line containing it becomes the new current line.

Viewing Program Data

When your program execution is suspended, you can look at the current values of its variables.

Listing the Variables

The **l** command gives you a listing of the values of all of the parameters and local variables in a particular routine. If **l** is typed by itself, the listing is for the current routine. If you enter

```
l routine
```

the listing is for the routine in your program called *routine*.

Finding a Variable's Value

You can find the value of a variable (or expression) with the `p` command:

```
p name
```

The debugger searches for a local variable or parameter called *name* in the current routine.

The following example shows the command used with an expression:

```
p 1+2
```

The response is 3.

To find the value of a local variable or parameter in a routine other than the current routine, type

```
p routine:name
```

which gives you the value of variable *name* during the most recent execution of *routine*. If the debugger cannot find a local variable or parameter called *name* in *routine*, the debugger looks for a common or static variable with that name.

Execution Control

There are two primary ways of controlling the execution of your program in the `xdb` environment:

- You can set breakpoints in the program that cause execution to be suspended at particular locations.
- You can use the `s` or `S` commands to single step through the program's execution.

Breakpoints

A breakpoint is a special debugger signal generator that can be inserted in a particular location in your program where you want execution to halt. Once you have halted the program, you can analyze its execution environment.

To use breakpoints, you should know:

- How to set them
- How to recover from them
- How to delete them

Setting Breakpoints

The command **b** sets breakpoints. It has many variations, but the simplest one has the form

b

This causes a breakpoint to be set at the current line of the current file and routine or the first executable statement following the current line. Using the commands mentioned earlier in “Viewing the Source File”, move to the location in your source file where you want the breakpoint. The file, routine, and line that you move to becomes the new current file, routine, and line. Type

v

to confirm what the current values are. To set the breakpoint, type

b

Another form of the **b** command allows you to specify any line in the current file where you want a breakpoint set. It has the form

b *n*

where *n* is the number of the line.

Recovering from Breakpoints

Breakpoints suspend program execution at particular locations. Once a program is suspended, you can resume execution at the place where it stopped with the **c** command.

Typing

c

causes the program to continue executing at the first executable statement following the statement that caused its suspension.

Deleting Breakpoints

To delete breakpoints from your program, use the `db` or `db*` command. The `db` command removes one breakpoint, while `db*` removes all breakpoints. By typing

```
db
```

you remove any breakpoint set at the current line. If there is no breakpoint to remove, you receive a listing of all of the breakpoints in your program. The lines of the listing are in the form

```
number routine:line count commands
```

number is an integer label that the debugger assigns to each breakpoint. *routine* and *line* locate the breakpoint by routine name and line number. *count* and *commands* are attributes of breakpoints that are described in the *HP Symbolic Debugger/iX User's Guide*.

To request a listing of the breakpoints in your program, type

```
lb
```

To delete a breakpoint at a location other than the current line, reference the breakpoint by the integer label *number* the debugger has assigned to it. First, use the `lb` command to get a list of the breakpoints and locate the one you want to delete. Next, use the *number* label associated with the breakpoint together with the `d` command in the form

```
d number
```

If you want to delete all of the breakpoints in your program, type

```
db*
```

Using Breakpoints for Execution Tracing

One variation of the set breakpoint command is useful when you want to be notified when a routine is called but do not want execution to halt. The command

```
bp routine
```

causes *routine*'s name to be printed when *routine* is called and then resumes execution.

Single Step Commands

The commands for specifying single step execution of your program are the `s` and `S` commands. Typing

`s`

executes one statement and suspends the program. Typing

`s n`

executes n statements before the program suspends.

The default value for n is 1.

The `S` command also allows single stepping through a program. When routine calls are reached in the currently executing routine, they are treated as statements and are executed. If, however, a called routine contains a breakpoint, execution halts at that point. As with the `s` command, you can specify the number of statements (n) to be executed.

To leave single step execution, use the `c` command. Typing

`c`

causes normal execution to resume.

Additional Debugger Capabilities

This chapter has given you a flavor of what the symbolic debugger can do and how you use it; however, there is much more available. Capabilities that are not covered here are:

- The *record/playback* mechanism. Record mode allows you to record a sequence of debugger commands that are required to get a program into a particular state. You can then use the playback mode to return the program to that state at any time.
- *Assertion control* commands. These commands are used to specify sets of commands that are executed after *every* statement in your program.
- Toggle disassembler/toggle source. This mode provides detailed information by displaying the machine instructions and registers used in the program's execution.

For more information on these capabilities, refer to the *HP Symbolic Debugger/iX User's Guide*.

Removing Debugging Information

Once your program executes correctly, remove the debugging code that has been incorporated into your program. The maintenance of the three tables used by the symbolic debugger increases the amount of space required by the program and slows down its linking speed. In addition, the tables supply many debugging capabilities that are not necessary (and not wanted) in the program's production version.

To remove the debugging code, you must recompile and relink the program without the `SYMDEBUG XDB` directive.

Using HP Toolset/iX

HP Toolset/iX performs the following:

- Sets breakpoints
- Interactively displays and modifies the values of variables
- Displays subroutines or functions currently called
- Traces each subroutine or function as it executes
- Traces variables as they are modified by the program

Compiling Programs for HP Toolset/iX

To symbolically debug, you must compile your program with the SYMDEBUG compiler directive, which instructs the compiler to generate additional information needed by HP Toolset/iX.

The directive

```
$SYMDEBUG [ ON ]
```

or

```
$SYMDEBUG [ TOOLSET ]
```

must be included at the beginning of your source file or passed to the compiler through the INFO string.

Invoking HP Toolset/iX

To invoke HP Toolset/iX, enter

```
RUN TOOLSET.PUB.SYS
```

or

```
TOOLSET
```

In the examples in this chapter, underlined items represent user input.

Setting Up for Symbolic Debug

Once you are in HP Toolset/iX, you must set up a workspace as shown below:

```
workspace wsname  
Create workspace wsname? yes  
Default language for wsname? fortran
```

A screen is displayed after the workspace is set up. The HP Toolset/iX workspace features described in the *HP Toolset/iX Reference Manual* are not available for HP FORTRAN 77/iX. To directly proceed to symbolic debug, press the f3 key (Set OK). This exits the “Set” screen and places you at the HP Toolset/iX prompt, “>>”.

Refer to the *HP Toolset/iX Reference Manual* for a detailed description of HP Toolset/iX.

When to Use HP Toolset/iX

Using the symbolic debugger increases the size of your program (often by a factor of two or three).

When your program executes correctly and you no longer need the symbolic debugger, remove the compiler option, recompile the source files, and relink the object files. Without the debugging information, the program occupies less space, allowing it to link and execute faster.

You might want to maintain two versions of a program: one with debugging information for support purposes and one production version that has the debugging information removed.

Running a Program

The RUN command executes a program that has been compiled with Symbolic Debug. The RUN command either executes the program that is currently set up in your workspace (which is useful for non-FORTRAN programs), or executes a program that you specify (used for FORTRAN programs). The program must have been compiled with the SYMDEBUG compiler directive. Refer to the *HP Toolset/iX Reference Manual* for the complete syntax of the RUN command and a detailed description of the parameters.

Syntax

```
RUN progfile
```

To make a program known to HP Toolset/iX, enter the following:

```
USE progfile
```

```
RUN progfile
```

Setting Breakpoints

The AT command allows you to set up to 15 active breakpoints in your program. You can set a breakpoint specifying a subroutine or function name or you can specify a location with an offset. The breakpoint is permanent unless the FOR clause is used. There are additional features of this command, as shown in the syntax below:

Syntax

$$\text{AT } \left\{ \begin{array}{l} \textit{location} \\ \text{NEXT} \end{array} \right\} \left[\left[\text{EVERY} \right] n \left[\text{TIMES} \right] \right] \left[\text{FOR } \left\{ \begin{array}{l} n \\ \text{ALL} \end{array} \right\} \left[\text{TIMES} \right] \right]$$
$$\left[\text{DO } \left[\begin{array}{l} \textit{command-list} \\ \text{BREAK} \end{array} \right] \right]$$

Examples

The following statement sets a breakpoint at the beginning of the subroutine named `sub1`:

```
AT sub1
```

The following statement sets a breakpoint at line 20 in the function named `func2`:

AT func2#20

Note



You cannot directly set a breakpoint at FORTRAN entry statements. However, using the AT NEXT command stops your program from entering any subroutine. Therefore, the AT NEXT command allows you to stop at entry statements.

Tracing Names

The CALLS command displays the names of the subroutines and functions that are currently executing. The trace begins with the most recently called subroutine or function. The statement following the call is also displayed.

Syntax

CA[LLS]

Note



For entry calls, the name of the enclosing subroutine is displayed. However, the accompanying statement number reflects execution of the entry statement.

Clearing Breakpoints

The CLEAR command removes breakpoints from your program.

Syntax

$$\text{CL}[\text{EAR}] \left\{ \begin{array}{l} \textit{location} \\ \textit{next} \\ \text{ALL} \end{array} \right\}$$

Example

The following statement clears the breakpoint at the location `sub1`:

```
CL sub1
```

Displaying Variables

The DISPLAY command causes the current contents of the specified FORTRAN variable (data item) to be displayed on your terminal in octal, integer, character, or hexadecimal format. By default, the variables are displayed in the format most appropriate for the data type.

Syntax

$$\text{DI}[\text{SPLAY}] \left\{ \begin{array}{l} \textit{rec-item} \\ \textit{data-item} \\ \textit{literal} \end{array} \right\} \left[\begin{array}{l} \text{O}[\text{CTAL}] \\ \text{I}[\text{NTEGER}] \\ \text{C}[\text{HARACTER}] \\ \text{H}[\text{EXADECIMAL}] \end{array} \right] \left[[\text{FOR}] n [\text{ITEMS}] \right]$$

Example

If your program has the declarations

```
INTEGER*4      long_integer_var
REAL*4         real_var
DOUBLE COMPLEX double_complex_var
LOGICAL*2      short_logical_var
INTEGER*4      array_var(2:5,1:3)
CHARACTER*8    array_char_var(4,4,4)
INTEGER*2      i, j
COMMON         /BLKA/i(4), j(6), alpha
CHARACTER*4    alpha
COMMON         k,l
```

The following is output using DISPLAY commands:

```
>>DISPLAY long_integer_var
--> Stmt #101: Var: LONG_INTEGER_VAR = 1234567890

>>DIS real_var
--> Stmt #101: Var: REAL_VAR = 0.123456E38

>>DIS double_complex_var
--> Stmt #101: Var: DOUBLE_COMPLEX_VAR = (8.2E308,7.1E308)

>>DIS short_logical_var
--> Stmt #101: Var: SHORT_LOGICAL_VAR = .TRUE.

>>DISPLAY array_var
--> Stmt #101: Var ARRAY_VAR
--> Starting with ARRAY_VAR[2:1]
4321 6654 87654321 6789
--> Continuing with ARRAY_VAR[2:2]
65432 121 159753 0
--> Continuing with ARRAY_VAR[2:3]
456 1 369 5
```

If the value of ARRAY_CHAR_VAR(1,2,3) is abcdefgh, the output is

```
>>DISPLAY array_char_var(1,2,3)FOR 4 ITEMS
--> Stmt #101: Var ARRAY_CHAR_VAR = 'efgh'
```

where the final subscript in the command is the start character.

```
>>DIS blka
--> Stmt #101: COMMON BLOCK: BLKA
I - ARRAY
--> Starting with I(1)
  0 4321 54 3
--> Starting with J(1)
  1 23 45 67 87 5
ALPHA = 'abcd'
```

Use the identifier COM' to display an unnamed common block. For example,

```
>>DIS COM'
--> Stmt #103: COMMON BLOCK: COM'
K=2
L=3
```

Modifying Variables

The MOVE command transfers the value of a literal, a figurative constant (such as .TRUE. and .FALSE.), or a variable, to another variable.

Syntax

$$\text{MOV[E] } \left\{ \begin{array}{l} \textit{literal} \\ \textit{data-item-1} \\ \textit{fig-constant} \end{array} \right\} \text{ TO } \textit{data-item-2} \text{ [[FOR] } n \text{ [ITEMS]]}$$

Example

If your program has the declarations

```
INTEGER*4      long_integer_var
REAL*4         real_var
REAL*8         long_real_var
DOUBLE COMPLEX double_complex_var
LOGICAL*2      short_logical_var
LOGICAL*4      array_var(25)
CHARACTER*8    array_char_var
```

the following is output using the MOVE command:

```
>>MOVE 45 to long_integer_var
--> Stmt#1: Var: LONG_INTEGER_VAR = 45

>>MO 3.14E38 TO real_var
--> Stmt#1: Var: REAL_VAR = 3.14E38

>>MO (13.3E73,14.4E60) TO double_complex_var
--> Stmt#1: Var: DOUBLE_COMPLEX_VAR = (13.3E73,14.4E60)

>>MO .TRUE. TO short_logical_var
--> Stmt#1: Var: SHORT_LOGICAL_VAR = .TRUE.

>>MO 20 TO array_var(1) FOR 5 TIMES
--> Stmt#1: Var: ARRAY_VAR[1]
--> Starting with ARRAY_VAR[1] 20 20 20 20 20

>>MO 'This is nice' TO array_char_var
--> Stmt#1: Var: ARRAY_CHAR_VAR = 'This is nice'

>>MO real_var TO long_real_var
--> Stmt#1: Var: long_real_var = 3.14E38
```

Redoing a Command

The REDO command allows you to correct and re-execute the last command or command list. The command editing is performed with EDITOR program operators.

Syntax

```
RED [ 0 ]
```

Example

```
>>MVE 30 TO long_integer_var
**Undefined TOOLSET/iX command keyword. (101)

>>REDO
MVE 30 TO long_integer_var
  i0
MOVE 30 TO long_integer_var
(CR)
-->Stmnt#1: Var: LONG_INTEGER_VAR = 30
```

Restarting Your Program

The RESUME command starts or restarts the execution of your program at the location at which it was halted.

Syntax

```
RES [ UME ]
```

Displaying Breakpoints

The SHOW DEBUG command displays the breakpoints and the names of any data item traced in the current program.

Syntax

```
SHO [ W ] D [ EBUG ]
```

Using the Trace Facilities

The TRACE command displays each subroutine or function name as it is executed. An identifying message is displayed at the start and end of each subroutine or function.

Syntax

```
T[RACE][OFF]
```

The DATATRACE command monitors the value of a data item. If the value changes, a message containing the location and the new value is displayed.

Syntax

```
DA[TATRACE] data-item [DO command-list [NOMESSAGE]]  
OFF
```

The RETRACE command lists the last *n* subroutines or functions that have executed, ending with the most recent subroutine or function.

Syntax

```
RET[RACE][PROCEDURES]
```

Accessing MPE/iX Debug

The SYSDEBUG command allows you to access MPE/iX DEBUG from HP Toolset/iX.

Syntax

```
SYS[DEBUG]
```

Ending Execution of a Program Prematurely

The END RUN command ends execution of your program prematurely. This command immediately terminates your program and returns control to HP Toolset/iX.

Syntax

```
EN[D]RU[N]
```

Exiting HP Toolset/iX

The EXIT command exits HP Toolset/iX and returns control to the MPE/iX operating system.

Syntax

```
EX[IT]
```

Example

```
>>EXIT  
END OF PROGRAM
```


Index

- 9** 900 Series HP 3000 architecture, 1-11

- A** ACCESS='DIRECT', 4-6
 - accessing disk, 3-1
 - accessing MPE/iX debug, 9-20
 - accessing the representation of logical values, 7-7
 - ACCESS='SEQUENTIAL', 4-5
 - ACCESS='SEQUENTIAL' option, 3-2
 - actual arguments
 - in subprograms, 5-18
 - in subroutines, 5-4
 - actual parameters, 8-21
 - addressing mode, 1-3
 - adjustable dimensions, 5-28
 - A format descriptor, 2-21
 - ALIAS compiler directive, 8-1
 - alignment of data, 1-2
 - ALIGNMENT option, 7-21
 - alternate returns from a subroutine, 5-5
 - ANSI compiler directive, 7-2
 - ANSI standard, 4-9
 - AOPTIONS, 4-1, 4-4
 - apostrophe edit descriptor, 2-43
 - apostrophes for character data, 2-2
 - appending to a file, 3-11
 - architecture, 1-11
 - argument lists, 5-18
 - arguments to subprograms, 5-18
 - arithmetic operators, 6-4
 - array elements, equivalence, 1-4
 - arrays, 2-57
 - adjustable dimensions, 5-28
 - assumed-size, 5-30
 - passed in a subprogram, 5-27
 - ASCII logical records, 4-9
 - assembly language routines, 6-16
 - assigning data areas, 1-12
 - ASSIGN statement, 2-55
 - assumed-size arrays, 5-30
 - ASSUME_NO_EXTERNAL_PARMS option, 6-29
 - ASSUME_NO_PARAMETER_OVERLAPS option, 6-27
 - ASSUME_NO_PARM_TYPES_MATCHED option, 6-28
 - ASSUME_NO_SIDE_EFFECTS option, 6-28
 - AT command, 9-14

- B**
 - BACKSPACE statement, 3-29, 4-5, 4-13
 - blank common blocks, 5-36
 - blanks in the input field
 - BN descriptor, 2-52
 - BZ descriptor, 2-52
 - BLANK specifier, 3-13
 - BLOCK DATA statement, 5-41
 - block data subprograms, 5-41
 - blocking factor, 4-9
 - BN edit descriptor, 2-52
 - breakpoints
 - clearing, 9-16
 - displaying, 9-19
 - setting, 9-14
 - bytes, returning number of, 2-49
 - BZ edit descriptor, 2-54

- C**
 - Calling HP COBOL II/iX from HP FORTRAN 77/iX, 8-10
 - Calling HP FORTRAN 77/iX from HP COBOL II/iX, 8-11
 - Calling HP FORTRAN 77/iX from HP Pascal/iX, 8-5
 - Calling HP Pascal/iX from HP FORTRAN 77/iX, 8-3
 - CALLS command, 9-15
 - CALL statement
 - alternate return, 5-5
 - invoking subroutines, 5-4
 - carriage control files, 4-8
 - CCTL, 4-8
 - CCTL file, 4-8
 - character data, 8-1
 - in a subprogram, 5-25
 - list-directed, 2-2
 - character format descriptors
 - A, 2-21
 - input field, 2-25
 - output field, 2-26
 - R, 2-21
 - character format descriptors and numeric data, 2-29
 - character positions
 - T edit descriptor, 2-40
 - TL edit descriptor, 2-41
 - TR edit descriptor, 2-42
 - X edit descriptor, 2-39
 - character variables, equivalence, 1-8
 - CHECK_ACTUAL_PARM compiler option, 8-1
 - CHECK_FORMAL_PARM compiler directive, 7-16, 8-1
 - CHECK_OVERFLOW directive, 6-8
 - CLEAR command, 9-16
 - clearing breakpoints, 9-16
 - CLOSE statement, 4-7, 4-13
 - description, 3-1
 - STATUS specifier, 3-3
 - closing files, 4-7
 - code area, 1-11
 - code space efficiency, 6-9

- colon edit descriptor, 2-50
- comments, 7-16
- common blocks, 1-12
 - blank common, 5-36
 - declaring, 7-5
 - description, 5-36
 - EQUIVALENCE statement, 1-9
 - labeled common, 5-39
- common blocks in memory, 1-12
- COMMON statement, 5-36
- common subexpression elimination module, 6-21
- compiler directive
 - HP3000_16, 7-19
 - STANDARD_LEVEL, 7-15
- compiler directives
 - INCLUDE, 7-5
 - LONG, 7-3
 - NOSTANDARD, 2-29
 - SHORT, 7-3
- compiler library, 4-7
- compile-time efficiency, 6-2
- complex format descriptors, 2-17
- conditional compiler directives, 7-17
- connecting files, 3-1
- constants passed in a subprogram, 5-21
- correcting a command, 9-19
- creating a new file, 3-6
- creating files, 4-4
- current record, 3-1, 3-12

D

- data alignment, 1-2, 1-10, 7-21
- data area, 1-11, 6-14
- data classes, 1-3
- data objects, 1-11
- data space efficiency, 6-10
- DATA statements, 6-8
- data storage
 - consistent, 7-3
 - description, 1-1
- DATATRACE command, 9-20
- data types, 8-2
- debug, 9-20
- debugging, 9-1
 - using xdb, 9-1
- default file properties, 3-2
- descriptor mode of addressing, 1-3
- descriptors, format, 2-10
- device type, 4-8
- D format descriptor, 2-17
- dimensions
 - adjustable, 5-28
- direct access file
 - creating, 3-15
 - description, 3-14

- reading and writing, 3-15
- direct access option, 4-6
- directives
 - OPTIMIZE, 6-25
- direct mode of addressing, 1-3
- disconnecting files, 3-1
- disk files, 3-1, 4-8
- DISPLAY command, 9-16
- displaying breakpoints, 9-19
- displaying functions, 9-20
- displaying names, 9-15
- displaying subroutines, 9-20
- displaying variables, 9-16
- DO loop, implied, 2-56
- DO loops, 6-8
- DO WHILE loops, 6-8
- dummy arguments
 - in subprograms, 5-18
 - in subroutines, 5-2

E

- \$ edit descriptor, 2-38
- / edit descriptor, 2-38
- : edit descriptor, 2-50
- edit descriptor
 - \$, 2-38
 - ;, 2-50
 - apostrophe, 2-43
 - BN, 2-52
 - BZ, 2-54
 - colon, 2-50
 - H, 2-44
 - NL, 2-38
 - NN, 2-38
 - P, 2-44
 - Q, 2-49
 - quotation mark, 2-43
 - S, 2-49
 - slash, 2-38
 - SP, 2-49
 - SS, 2-49
 - summary, 2-11
 - T, 2-40
 - TL, 2-41
 - TR, 2-42
 - X, 2-39
- efficiency
 - code space, 6-9
 - compile-time, 6-2
 - data space, 6-10
 - run-time, 6-3
- efficient programs, 6-1
- E format descriptor, 2-17
- ELSE compiler directive, 7-17
- ENDFILE statement, 3-29, 4-13

- ENDIF compiler directive, 7-17
- ending execution, 9-20
- end-of-file record, 3-12, 3-34
- END RUN command, 9-20
- END statement, 4-7
 - description, 5-10
 - subroutines, 5-3
- entries into subprograms, 5-32
- ENTRY statement, 5-32
- EOF, 4-8
- equivalence
 - array elements, 1-4
 - character variables, 1-8
- EQUIVALENCE statement
 - array elements, 1-4
 - arrays with different dimensions, 1-7
 - avoid using, 7-5
 - character variables, 1-8
 - common blocks, 1-9
 - data alignment, 1-10
 - data storage, 1-4
 - description, 1-4
 - multi-dimensioned arrays, 1-6
- errors, file handling, 3-2
- error termination, 4-7
- ERR specifier, 3-3
- ERR specifiers, 4-7
- examples
 - file handling, 3-31
 - using file positioning statements, 3-30
- EXIT command, 9-20
- exiting HP Toolset/iX, 9-20
- expressions passed in a subprogram, 5-22
- extents, 4-1

F

- FCLOSE, 4-7
- FCONTROL intrinsic, 4-5
- F format descriptor, 2-17
- file
 - closing, 4-7
- file access
 - description, 3-12
 - direct, 3-14
 - indexed sequential, 3-18
 - ISAM, 3-18
 - sequential, 3-12
- file characteristics, 4-1
- FILE command, 4-1
- file connections, 4-1
- FILE equation, 4-1, 4-8, 4-9
- file handling errors, 3-2
 - ERR specifier, 3-3
 - IOSTAT specifier, 3-5
 - STATUS specifier, 3-3

- file handling procedures, 4-10
- file handling statement examples, 3-31
- file operations, 4-1
- file pointer, 3-1, 3-29
- file positioning
 - BACKSPACE statement, 3-29
 - ENDFILE statement, 3-29
 - examples, 3-30
 - REWIND statement, 3-29
- file properties, 3-2
- files
 - appending to, 3-11
 - creating, 3-6
 - direct access, 3-14
 - formatted, 3-24
 - indexed sequential, 3-18
 - INQUIRE statement, 3-26
 - internal, 3-34
 - ISAM, 3-18
 - reading, 3-9
 - sequential, 3-12
 - unformatted, 3-24
- files,creating, 4-4
- file size, 4-1
- FILESIZE, 4-1
- files,predefined, 4-2
- FNUM procedure, 4-12
- FOPEN intrinsic, 4-1
- FOPTIONS, 4-1
- FOR clause, 9-14
- formal parameters, 8-21
- format control, 2-50
- @ format descriptor, 2-12
- format descriptor
 - character, 2-29
 - monetary, 2-34
 - Mw.d, 2-34
 - numeration, 2-36
 - Nw.d, 2-36
 - repeating, 2-31
- format specifications, 2-1, 2-12
- FORMAT statement, 2-5
- formatted files, 3-24
- formatted form option, 4-5
- formatted input, 2-6
- formatted input/output, 2-1
- formatted output, 2-7
- formatted statements, 2-5
- FORM='FORMATTED', 4-5
- FORM='FORMATTED' option, 3-2
- FORM='UNFORMATTED', 4-5
- FORTRAN/3000, 4-9
- FORTRAN 77 library, 4-7
- FREADDIR, 4-5
- free format, 2-1

- FSET procedure, 4-10
- FTN05, 4-2
- FTN06, 4-2
- functions, 5-1
 - categories, 5-9
 - description, 5-9
 - intrinsic, 5-16
 - statement, 5-15
 - subprograms, 5-9
 - user-defined, 5-10
- FUNCTION statement, 5-9, 5-12
- FWRITE, 4-8
- FWRITEDIR, 4-5

G G format descriptor, 2-17
GIVING phrase, 8-11
grouping related routines, 6-12

H H edit descriptor, 2-44
hollerith edit descriptor, 2-44
HP3000_16 compiler directive, 6-8

- description, 7-19
- options, 7-20

HP COBOL II/iX, 8-8
HP Pascal/iX, 8-2
HP Toolset/iX

- description, 9-1
- exiting, 9-20
- invoking, 9-13
- when to use, 9-14

I IF compiler directive, 7-17
I format descriptor, 2-12
IF statements, 6-6
IMPLICIT NONE statement, 7-5
implied DO loop, 2-56
improving MPE/iX run-time efficiency, 6-8
INCLUDE compiler directive, 7-5
INCLUDE statement, 7-5
inconsistencies of data storage, 7-16
indexed sequential access files, 3-18
indirect mode of addressing, 1-3
induction variables, 6-20
initialized variables, 1-12
initializing data, 7-7
input

- unformatted, 3-23

input/output statement specification, 2-55
INQUIRE statement, 3-26, 4-6
integer format descriptors

- @ descriptor, 2-12
- I descriptor, 2-12
- input field, 2-13
- K descriptor, 2-12

- O descriptor, 2-12
- output field, 2-15
- Z descriptor, 2-12
- interfacing with other languages, 8-1
- internal files
 - description, 3-34
 - reading, 3-34
 - writing, 3-36
- intrinsic functions, 5-16
- intrinsic I/O, 6-8
- intrinsic,system, 8-20
- invoking subroutines, 5-4
- I/O errors, 4-7
- I/O library, 4-8
- IOSTAT specifier, 3-5
- IOSTAT specifiers, 4-7
- ISAM, 3-18

K K format descriptor, 2-12

L labeled common blocks, 5-39

- languages,interfacing with, 8-1
- length specifications, 7-4
- L format descriptor, 2-30
- line printer, 4-8
- link editor, 1-12, 8-1
- list-directed
 - character data, 2-2
 - input, 2-1
 - output, 2-3
 - READ statement, 2-1
- list-directed I/O, 2-1
- literal data, 2-43
 - apostrophe edit descriptor, 2-43
 - H edit descriptor, 2-44
 - quotation mark edit descriptor, 2-43
- LOCALITY compiler directive, 6-8
- local variables in memory, 1-12
- logical format descriptor
 - input field, 2-30
 - L descriptor, 2-30
 - output field, 2-30
- logical records, 4-9
- logical values, 7-7
- LONG compiler directive, 7-3

- M** machine instructions, 1-11
- magnetic tapes, 4-9
- main memory, 6-12
- maintaining parameter type and length consistency, 7-7
- memory area assignment, 1-12
- memory areas, 1-11
- memory areas, summary, 1-13
- memory data areas, 3-34
- modifiable programs, 7-9
- modifying variables, 9-18
- monetary data field, 2-34
- monetary format descriptor, 2-34
- MOVE command, 9-18
- MPE/iX debug, 9-20
- MPE/iX operating system, 4-7
- MPE/iX run-time efficiency, 6-8
- MPE V operating system, 7-19
- MPE V system, 7-22
- multiple entries into subprograms, 5-32
- Mw.d format descriptor, 2-34

- N** named common blocks, 1-12
- new files, 3-6
- new lines, 2-38
 - \$ descriptor, 2-38
 - / descriptor, 2-38
 - NL descriptor, 2-38
 - NN descriptor, 2-38
 - slash descriptor, 2-38
- new status option, 4-4
- NL edit descriptor, 2-38
- NN edit descriptor, 2-38
- noncharacter data, 7-21
- nonstandard features, 7-15
- normal termination, 4-7
- NOSTANDARD compiler directive, 2-29
- number of bytes, Q edit descriptor, 2-49
- number of extents, 4-1
- numeration data field, 2-36
- numeration format descriptor, 2-36
- numeric data types, 8-8
- NUMEXTENTS, 4-1
- Nw.d format descriptor, 2-36

- O**
 - OFF option, 7-20
 - O format descriptor, 2-12
 - OLD files, 4-4
 - old status option, 4-4
 - ON option, 7-20
 - OPEN statement
 - ACCESS='SEQUENTIAL' option, 3-2
 - appending to a file, 3-11
 - connecting files, 3-1
 - creating files, 4-4
 - description, 4-1
 - ERR specifier, 3-3
 - FORM='FORMATTED' option, 3-2
 - IOSTAT specifier, 3-5
 - options, 4-1
 - processor, 4-1
 - reporting errors, 3-2
 - STATUS='OLD' status, 3-9
 - STATUS specifier, 3-3
 - STATUS='UNKNOWN' option, 3-2
 - optimization
 - techniques, 6-1
 - optimization, branch, 6-17
 - optimization, level one, 6-15
 - optimization, level one, 6-15
 - optimization, level one modules, 6-16
 - optimization, level two, 6-15, 6-19
 - optimization, level two, 6-15
 - OPTIMIZE compiler directive, 6-15
 - OPTIMIZE directive, 6-25
 - optimized programs, troubleshooting, 6-38
 - OPTIMIZE options, 6-25
 - optimizer assumptions, 6-23
 - output
 - unformatted, 3-23
 - overlapping character substrings, 7-25

- P**
 - parameter list, 8-1
 - PARAMETER statement, 2-55
 - parameter types, 7-7
 - Pascal, 8-2
 - Pascal data types, 8-21
 - passing arrays, 5-27
 - passing by reference, 8-1
 - passing by value, 8-1
 - passing character data, 5-25
 - passing constants, 5-21
 - passing expressions, 5-22
 - passing subprograms, 5-31
 - P edit descriptor, 2-44
 - performance tuning, 6-11
 - plus sign
 - S edit descriptor, 2-49
 - SP edit descriptor, 2-49

- SS edit descriptor, 2-49
- portable programs, 7-1
- positioning the file pointer, 3-29
- preconnected units, 2-1
- predefined files, 4-2
- predefined units, 4-2
- prespacing mode, 4-8
- PRINT statement
 - formatted, 2-7
- programming for portability, 7-1
- program,terminating, 4-7
- program unit, 5-5
- proprietary floating point data, 7-20

Q Q edit descriptor, 2-49
quotation mark edit descriptor, 2-43

R reading an existing file, 3-9
READ statement, 6-8

- formatted input, 2-6
- list-directed, 2-1
- unformatted, 3-23

real format descriptors, 2-17

- D descriptor, 2-17
- E descriptor, 2-17
- F descriptor, 2-17
- G descriptor, 2-17
- input field, 2-17
- output field, 2-19

REALS option, 7-24
RECL option, 4-6
recursive subroutines, 5-2, 5-10
REDO command, 9-19
redoing a command, 9-19
removing breakpoints, 9-16
repeating specifications, 2-31
reporting file handling errors, 3-2
restarting a program, 9-19
restricting programs to HP FORTRAN 77 standard, 7-2
RESUME command, 9-19
RETRACE command, 9-20
RETURN statement

- alternate return, 5-5
- containing an expression, 5-5
- description, 5-10
- subroutines, 5-3

REWIND statement, 3-29, 4-13
R format descriptor, 2-21
routines, grouping related, 6-12
RUN command, 9-14
running a program, 9-14
run-time

- efficiency, 6-3

S saved variables in memory, 1-12
 SAVE statement, 1-12, 5-43
 scale factors, 2-44
 scratch status option, 4-4
 S edit descriptor, 2-49
 sequential access, 4-5
 sequential access files, 3-12
 SET compiler directive, 7-17
 setting breakpoints, 9-14
 setting up a workspace, 9-13
 setting up for symbolic debug, 9-13
 shifting data, 6-14
 SHORT compiler directive, 4-14, 7-3
 SHOW DEBUG command, 9-19
 slash edit descriptor, 2-38
 slash in input field, 2-2
 SP edit descriptor, 2-49
 SS edit descriptor, 2-49
 stack area, 1-11, 6-14
 standard features of HP FORTRAN 77, 7-2
 standard input, 2-1
 STANDARD_LEVEL compiler directive, 7-15
 standard output, 2-1
 starting a program, 9-19
 statement functions, 5-1, 5-15
 STATUS='NEW', 4-4
 STATUS='OLD', 4-4
 STATUS='SCRATCH', 4-4
 STATUS specifier, 4-7
 'DELETE' status, 3-3
 'KEEP' status, 3-3
 'NEW' status, 3-3, 3-6
 'OLD' option, 3-9
 'OLD' status, 3-3
 'SCRATCH' status, 3-3
 'UNKNOWN' status, 3-2, 3-3
 STATUS='UNKNOWN', 4-4
 \$STDINX, 4-2
 \$STDLIST, 4-2
 STOP statement, 4-7
 subroutines, 5-3
 storage allocation
 addressing mode, 1-3
 EQUIVALENCE statement, 1-4
 variable types, 1-2
 storage assignment, 1-11
 STRING_MOVE option, 7-25
 structure of a subroutine, 5-2
 subprograms
 arguments, 5-18
 block data, 5-41
 categories, 5-1
 multiple entries, 5-32
 passing, 5-31
 SAVE statement, 5-43

- subroutines
 - alternate returns, 5-5
 - common block, 5-36
 - description, 5-2
 - invoking, 5-4
 - recursive, 5-2
 - structure, 5-2
- SUBROUTINE statement, 5-2
- subsystems, 8-20
- summary of the memory areas, 1-13
- symbolic debugger
 - additional capabilities, 9-12
 - breakpoints, 9-9
 - controlling execution, 9-9
 - delete breakpoints, 9-11
 - description, 9-1
 - execute program, 9-5
 - execution control, 9-9
 - exiting, 9-5
 - invocation, 9-4
 - listing variables, 9-7
 - move command, 9-7
 - program requirement, 9-3
 - recover from breakpoints, 9-10
 - remove debugging information, 9-12
 - search commands, 9-7
 - set breakpoints, 9-10
 - single step execution, 9-12
 - tracing execution using breakpoints, 9-11
 - values of variables, 9-9
 - view command, 9-6
 - view execution stack, 9-5
 - view program data, 9-7
 - view source file, 9-6
 - window command, 9-7
- symbolic debugging, 9-1
- SYMDEBUG compiler directive, 9-14
- SYSDEBUG command, 9-20
- system intrinsics
 - defining, 8-20
 - description, 8-20

T TABLES ON compiler directive, 1-12

- tapes,magnetic, 4-9
- T edit descriptor, 2-40
- TEMP files, 4-4
- terminal printer, 4-8
- terminating a program, 4-7
- termination
 - error, 4-7
 - normal, 4-7
- TL edit descriptor, 2-41
- Toolset/iX
 - description, 9-1

- exiting, 9-20
- invoking, 9-13
- when to use, 9-14
- TRACE command, 9-20
- trace facilities
 - DATATRACE command, 9-20
 - RETRACE command, 9-20
 - TRACE command, 9-20
- tracing names, 9-15
- transferring values, 9-18
- transportable programs, 7-1
- TR edit descriptor, 2-42

U

- unformatted files, 3-24
- unformatted form option, 4-5
- unformatted input, 3-23
- unformatted I/O, 3-23
- unformatted output, 3-23
- unformatted READ statement, 3-23
- unformatted WRITE statement, 3-23
- uninitialized local variables, 1-12
- uninitialized variables, OPTIMIZE directive, 6-30
- UNITCONTROL intrinsic, 4-9
- UNITCONTROL options, 4-13
- UNITCONTROL procedure, 4-13
- unit numbers, 2-1
- units, predefined, 4-2
- unknown status option, 4-4
- unnamed common blocks, 1-12
- unstructured features, 7-15
- using consistent data storage, 7-3

V

- variable assignment, 1-12
- variable format descriptors, 2-8
- variables
 - displaying, 9-16
- variable types, 1-2

W

- workspace, 9-13
- workspace program file, 9-13
- WRITE statement, 6-8
 - formatted, 2-7, 3-24
 - list-directed output, 2-3
 - unformatted, 3-23
- writing efficient programs, 6-1

X

- xdb
 - commands, 9-2
 - description, 9-1
- X edit descriptor, 2-39

Z

- Z format descriptor, 2-12