HP FORTRAN 77

Programmer's Guide

HEWLETT-PACKARD COMPANY
Computer Languages Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

Part No. 5957-4686
Printed in U.S.A. March 1985
E0385

Printing History

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition .... Mar 1985

NOTICE

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

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

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

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 program language without the prior written consent of Hewlett-Packard Company.

Copyright © 1985 by HEWLETT-PACKARD COMPANY

PREFACE

The HP FORTRAN 77 Programmer's Guide contains a detailed discussion of selected FORTRAN topics. This manual is intended for an experienced programmer who has some familiarity with HP systems, data processing concepts, and the FORTRAN programming language. This manual does not discuss every feature of FORTRAN; for a complete discussion about HP FORTRAN 77, refer to the *HP FORTRAN 77 Reference Manual*.

This manual contains the following:

Chapter 1	describes the factors that influence the 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 disc and internal files.
Chapter 4	describes subroutine, function, and block data subprograms.
Chapter 5	describes all of the intrinsic functions, including the generic and specific functions.
Chapter 6	describes ways you can improve your program's efficiency.
Chapter 7	describes techniques to move programs from other systems onto HP systems, and how to make new programs easily transportable between HP systems.
Appendix A	describes HP FORTRAN 77 system-dependent information on the HP 3000.
Index	is a cross-reference index of all topics covered in this manual.

Additional Documentation

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

- *HP FORTRAN 77 Reference Manual* (Part number 5957-4685)

  This manual is a complete reference on all HP FORTRAN 77 features.

- *HP FORTRAN 77 Self Study Guide* (Part number 22999-90548)

  This manual provides tutorial instruction on fundamental programming concepts and features of HP FORTRAN 77.

- *Supplement to the HP FORTRAN 77 Self Study Guide* (Part number 22999-90549)

  This manual supplies machine-specific information on the HP 9000 Series 500 HP-UX operating system, HP 1000 RTE-6/VM and RTE-A operating systems, and HP 3000 MPE operating system.

- *MPE Debug/Stack Dump Reference Manual* (Part number 30000-90012)

  This manual describes the debug commands for the HP 3000.

- *MPE Intrinsics Reference Manual* (Part number 30000-90010)

  This manual describes the system intrinsics in detail.

Table of Contents

Chapter 1 Page
DATA STORAGE

Variable Types 1-1
Addressing Mode 1-1
The EQUIVALENCE Statement 1-2
Equivalence of Array Elements 1-3
Equivalence Between Arrays of Different Dimensions 1-4
Equivalence of Character Variables . . 1-5
Equivalence in Common Blocks 1-6
Equivalence and Data Alignment . . . 1-7

Chapter 2 Page
FORMATTED INPUT/OUTPUT

List-Directed Statements 2-1
List-Directed Input 2-1
List-Directed Output 2-2
Formatted Statements 2-4
Formatted Input 2-4
Formatted Output 2-6
Summary of the Descriptors 2-7
Format Specifications 2-8
Integer Format Descriptors: I, O, K, @, Z 2-8
The Input Field 2-9
The Output Field 2-11
Real Format Descriptors: F, D, E, G 2-12
The Input Field 2-12
The Output Field 2-13
Character Format Descriptors: A, R 2-15
The Input Field 2-17
The Output Field 2-18
Logical Format Descriptor: L 2-20
The Input Field 2-21
The Output Field 2-21
Repeating Specifications 2-22
Correspondence Between the I/O List and Format Descriptors . . . 2-22
Processing New Lines 2-24
The / Descriptor 2-24
The NL, NN, or $ Descriptor . . . 2-25
Handling Character Positions 2-25
The X Descriptor 2-25
The T Descriptor 2-26
The TL Descriptor 2-27
The TR Descriptor 2-28
Handling Literal Data 2-28
The ' and " Descriptors 2-28
The H Descriptor 2-29
Using Scale Factors: The P Descriptor 2-30
Printing Plus Signs: The S, SP, and SS Descriptors 2-33
Returning the Number of Bytes: The Q Descriptor 2-34
Terminating Format Control: The Colon Descriptor 2-34
Handling Blanks in the Input Field . 2-36
The BN Descriptor 2-36
The BZ Descriptor 2-37
Alternative Methods of Specifying Input/Output 2-38
Using the Implied DO Loop 2-39

Chapter 3 Page
FILE HANDLING

Disc 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 3-2
The ERR Specifier 3-4
The IOSTAT Specifier 3-4
Creating a New File Using STATUS='NEW' 3-4
Reading from an Existing File Using STATUS='OLD' 3-7
Appending to a File 3-9

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Table of Contents (Cont.)

File Access 3-11
Sequential Access Files 3-11
Direct Access Files 3-12
Unformatted I/O 3-15
Unformatted Input 3-15
Unformatted Output 3-16
Using Formatted and Unformatted Files 3-16
Using the INQUIRE Statement . . . 3-18
Positioning the File Pointer 3-20
The BACKSPACE Statement . . . 3-20
The REWIND Statement 3-20
The ENDFILE Statement 3-20
Example of Using the File Positioning Statements 3-21
File Handling Examples 3-22
Computing the Mean of Data In a Sequential File 3-22
Inserting Data Into a Sorted Sequential File 3-23
Internal Files 3-24
Reading from an Internal File 3-24
Writing to an Internal File 3-26

Chapter 4 Page
SUBPROGRAMS

Subroutines 4-2
Structure of a Subroutine 4-2
Invoking a Subroutine 4-3
Alternate Returns From a Subroutine 4-4
Functions 4-7
Function Subprograms 4-7
Statement Functions 4-13
Intrinsic Functions 4-15
Arguments to Subprograms 4-15
Passing Constants 4-17
Passing Expressions 4-18
Passing Character Data 4-20
Passing Arrays 4-21
Adjustable Dimensions 4-22
Assumed-Size Arrays 4-24
Passing Subprograms 4-25
Multiple Entries into Subprograms . . . 4-25
Common Blocks 4-28
Blank Common Blocks 4-28
Labeled Common Blocks 4-30
Block Data Subprograms 4-31
Using the SAVE Statement 4-33
Calling Subprograms Written in Other Languages 4-34

Chapter 5 Page
INTRINSIC FUNCTIONS

Invoking an Intrinsic Function 5-1
Generic and Specific Function Names 5-1
Summary of the Intrinsic Functions . . . 5-2
Function Descriptions 5-7
ABS 5-7
ACOS 5-8
ACOSH 5-8
AINT 5-9
ANINT 5-9
ASIN 5-10
ASINH 5-10
ATAN 5-11
ATANH 5-11
ATAN2 5-12
BTEST 5-12
CHAR 5-13
CMPLX 5-13
CONJG 5-14
COS 5-14
COSH 5-15
DBLE 5-15
DCMPLX 5-16
DIM 5-16
DPROD 5-17
EXP 5-17
IAND 5-18
IBCLR 5-18
IBITS 5-19
IBSET 5-19
ICHAR 5-20
IEOR 5-20
IMAG 5-21
INDEX 5-21
INT 5-22
IOR 5-22
ISHFT 5-23
ISHFTC 5-23

Table of Contents (Cont.)

IXOR 5-24
LEN 5-24
LGE 5-25
LGT 5-25
LLE 5-26
LLT 5-26
LOG 5-27
LOG10 5-27
MAX 5-28
MIN 5-28
MOD 5-29
MVBITS 5-29
NINT 5-31
NOT 5-31
REAL 5-32
SIGN 5-32
SIN 5-33
SINH 5-33
SQRT 5-34
TAN 5-34
TANH 5-35

Chapter 6 Page
WRITING EFFICIENT PROGRAMS

Compile Time Efficiency 6-1
Run-Time Efficiency 6-2
Declare Integer and Logical Variables Efficiently 6-2
Avoid Using Arrays 6-2
Use Efficient Data Types 6-2
Avoid Mixed-Mode Expressions 6-2
Eliminate Slow Arithmetic Operators 6-3
Use Statement Functions 6-3
Reduce External References 6-3
Combine DO Loops 6-4
Eliminate Short DO Loops 6-4
Eliminate Common Operations in Loops 6-5
Use Efficient IF Statements 6-5
Avoid Formatted I/O 6-6
Specify the Array Name for I/O 6-6
Avoid Using Range Checking 6-6
Use Your System Language 6-7
Minimize Segment Faults 6-7
Code Space Efficiency 6-7
Use Function Subroutines 6-7
Avoid Formatted I/O 6-7
Use Character Substrings 6-7
Data Space Efficiency 6-8
Eliminate Redundant or Unused Variables 6-8
Avoid COMMON Variables 6-8
Group Variables 6-8
Use Short Integer and Logical Data . . 6-8

Chapter 7 Page
PROGRAMMING FOR PORTABILITY

Restricting Programs to the HP FORTRAN 77 Standard 7-1
Using Consistent Data Storage 7-2
Use the LONG and SHORT Compiler Directives 7-2
Use Length Specifications in All Type Statements 7-3
Declare All Variables 7-3
Avoid Using the EQUIVALENCE Statement . . 7-3
Declare Common Blocks the Same in Every Program Unit . . 7-3
Initialize Data Before the Algorithm Begins 7-4
Avoid Accessing the Representation of Logical Values 7-4
Maintain Parameter Type and Length Consistency 7-4
Writing Programs that Can Be Easily Modified 7-5
Avoiding Unstructured FORTRAN 77 Features 7-10

Appendix A Page
USING FORTRAN 77 ON THE HP 3000

FORTRAN 77 File Operations on the HP 3000 . . . . A-1
Predefined Units and Files A-1
FTN05 . . . . A-2
FTN06 . . . . A-2
Creating Files with the OPEN Statement A-2
STATUS='NEW' A-2
STATUS='OLD' A-2
STATUS='SCRATCH' A-3
STATUS='UNKNOWN' A-3
FORM='UNFORMATTED' and FORM='FORMATTED' . . A-3
ACCESS='SEQUENTIAL' A-3
ACCESS='DIRECT' A-3

Table of Contents (Cont.)

Closing Files A-4
The CLOSE Statement A-4
Terminating the Program A-4
Carriage Control Files A-5
Terminals and Line Printers A-5
Disc Files A-5
:FILE Equation A-5
Using Magnetic Tapes A-5
Using the FSET Procedure A-6
Using the FNUM Procedure A-8
Using the UNITCONTROL Procedure A-8
Passing Run Command Parameters . . A-11
Accessing Data A-12
Debugging FORTRAN 77 Programs . . A-12
MPE Debug A-12
$RANGE Directive A-17
Writing Efficient Programs A-18
Programming for Portability A-19
Identify Nonstandard Features A-19
Avoid Inconsistencies A-19
Use Comments A-20
Use Conditional Compilation Directives A-20
Interfacing With Other Languages . . . A-21
SPL/3000 . . . . A-22
Calling SPL/3000 from FORTRAN 77 . . . . A-22
Calling FORTRAN 77 from SPL/3000 . . . . A-24
FORTRAN/3000 . . . . A-25
Calling FORTRAN/3000 from FORTRAN 77 . . . . A-25
Calling FORTRAN 77 from FORTRAN/3000 . . . . A-28
Pascal/3000 . . . . A-28
Calling Pascal from FORTRAN 77 . . . . A-28
Calling FORTRAN 77 from Pascal A-30
COBOL/3000 . . . . A-32
Using System Intrinsics A-34
Defining System Intrinsics A-35
Matching Actual and Formal Parameters A-35
Matching SPL and FORTRAN 77 Data Types A-37
Using HP 3000 Subsystems A-37
Sort-Merge A-37
Image A-40
VPLUS A-43
Stack Architecture A-46

List of Tables

Table 1-1. Addressing Mode 1-2
Table 1-2. Data Class and Addressing Mode 1-2
Table 2-1. Summary of the Format Descriptors 2-7
Table 2-2. Summary of the Edit Descriptors 2-8
Table 3-1. Status Types 3-3
Table 3-2. The STATUS Specifier 3-3
Table 3-3. The IOSTAT Specifier 3-4
Table 4-1. Components of Program Units 4-1
Table 5-1. Arithmetic Functions 5-3
Table 5-2. Bit Manipulation Functions 5-4
Table 5-3. Character Functions 5-4
Table 5-4. Numeric Conversion Functions 5-5
Table 5-5. Transcendental Functions . . 5-6
Table A-1. Summary of Accessing Modes A-12
Table A-2. Summary of Conditional Compilation Directives A-20
Table A-3. FORTRAN 77 and SPL/3000 Types A-23
Table A-4. FORTRAN 77 and FORTRAN/3000 Types A-26
Table A-5. FORTRAN 77 and PASCAL/3000 Types A-29
Table A-6. COBOL Numeric Types and Formats A-33
Table A-7. SPL and FORTRAN 77 Data Types A-36

Table of Contents (Cont.)

List of Figures

Figure 2-1. Differences Between the A and R Descriptors 2-19
Figure 3-1. Sequential Access Files . . . 3-11
Figure 3-2. Direct Access Files 3-12
Figure 5-1. The MVBITS Procedure . . 5-30
Figure A-1. Sample Code Segment Layout for FORTRAN 77 . . . . . . A-47
Figure A-2. Sample Data Stack Layout for FORTRAN 77 . . . . . . A-48

Chapter 1
Data Storage

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

- Variable types
- Addressing mode
- The EQUIVALENCE statement

This chapter describes these factors in detail.

Variable Types

FORTRAN data can be one of these types:

```
INTEGER
INTEGER*2
REAL
DOUBLE PRECISION
COMPLEX
DOUBLE COMPLEX
LOGICAL
LOGICAL*2
CHARACTER
```

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 with byte alignment is allocated seven bytes of storage, starting on a byte boundary. The size of each data type is the same on different systems, but the alignment requirement is usually system dependent. For more details of the format and alignment for each data type, refer to the *HP FORTRAN 77 Reference Manual*.

Addressing Mode

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

Table 1-1. 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 1-2.

Table 1-2. Data Class and Addressing Mode

DATA CLASS	ADDRESSING MODE
Common variables	Direct or indirect*
Static variables (variables in a SAVE or DATA statement)	Direct or indirect*
Variables in an EQUIVALENCE statement	Direct or indirect*
Parameters: Character string parameters Other parameters	 By descriptor Indirect
Other Variables: size $\leq n$ bytes** size $> n$ bytes**	 Direct Indirect

* System specific; see system specific appendix for details.

** *n* is system specific; see the system specific appendix for details.

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 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), and 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 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        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 i(5)
CHARACTER*16 c
EQUIVALENCE(i,c)
```

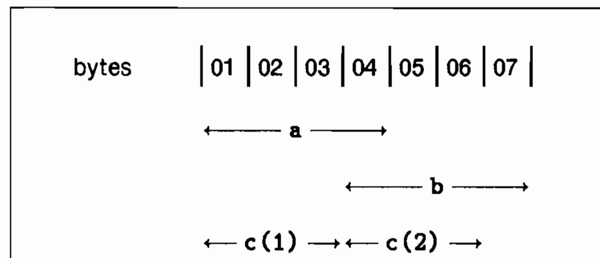
produce the following storage space allocation:

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	Unused

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 thus 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 following 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(2) is equivalent to i(3).

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

Array *i*	Storage Space Byte Number	Variable *j*
i(1)	1-4	
i(2)	5-8	j(1)
i(3)	9-12	j(2)
i(4)	13-16	j(3)
i(5)	17-20	j(4)
i(6)	21-24	j(5)
	25-28	j(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. Thus, the statement

```
EQUIVALENCE (i(1), j(2))
```

is not allowed.

Equivalence and Data Alignment

Each data type has its own data alignment requirement, which is system dependent; see the system-specific appendix in the *HP FORTRAN 77 Reference Manual* 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 boundaries and 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 j 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
Unused	0-1	1 unused byte, c(1:1)
j(1)	2-3	c(2:3)
j(2)	4-5	c(4:5)
Unused	6-6	c(6:6), 1 unused byte

Note that c starts on a byte boundary and j starts on an even byte (16-bit word) boundary, which may not be the actual machine word boundary.

Chapter 2
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 interactive devices (such as terminals), line printers, or disc 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 (hence 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 must be enclosed in apostrophes when read with a list-directed READ statement. This is because character data can contain blanks, which list-directed input uses for data separators. (Character data does not have to be enclosed in apostrophes when the input field is specified by format descriptors.) For example, the statements

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

accept the following input string:

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

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
```

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 illustrate the two ways of writing list-directed input and output statements.

```
      PROGRAM list_directed_io

*  List-directed input statements:

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

*  List-directed output statements:

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

      END
```

Here 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 1.23456E+31
```

NOTE

The exact output of floating point numbers might vary between systems. For example, the floating point number 1.0 may be output as 1. on some systems. Also, the number 1.23455E+31 might be output as 1.23456E+31, depending upon the accuracy of your machine.

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.

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 program below shows several ways of writing formatted input statements.

```
      PROGRAM formatted_input

*  Formatted input statements from the preconnected input unit:

      READ(5, '(I9)') i

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

*  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 expression 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 program below shows various ways of writing formatted output statements.

```
      PROGRAM formatted_output
      i = 15

*  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.

*  Formatted output statements to standard output device:

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

      PRINT '(1X, "i=", I9)', i
      PRINT 200, i                        ! Note: This statement reuses the
*                                           FORMAT statement labeled 200 above
      END
```

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, Aw R, Rw	All data types All data types
Logical	L	Lw	LOGICAL
Real	D E F G	Dw.d Ew.d, Ew.dEe Fw.d Gw.d, Gw.dEe	All numeric data types All numeric data types All numeric data types All numeric data types
Integer	I	Iw, Iw.m	All numeric data types
Octal	O, K, @	Ow, Kw, @w	Input: INTEGER Output: All data types
Hexadecimal	Z	Zw	Input: INTEGER Output: All data types

where:

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

NOTE

Wherever "LOGICAL" appears in Table 2-1, both LOGICAL*2 and LOGICAL*4 are applicable. Wherever "INTEGER" appears, both INTEGER*2 and INTEGER*4 are applicable. Wherever "all numeric data types" appears, INTEGER*2, INTEGER*4, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX*8, COMPLEX*16, and DOUBLE COMPLEX are applicable.

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.

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
	*n*H	Character	Outputs character strings
Input/Output	*n*P	Scale factor	Modifies output of the E*w.d*, D*w.d*, and G*w.d* descriptors and input of the F*w.d* descriptor
	*n*X	Position edit	Skips *n* positions
	T*n*	Tab edit	Positions to column *n*
	TL*n*	Tab edit	Positions backward *n* columns
	TR*n*	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 system specific appendix in the *HP FORTRAN 77 Reference Manual* for details.

where:

n is a positive, nonzero integer

The input descriptors are ignored on output. The output descriptors are ignored on input, except that *n*H, ", and ' are treated as *n*X, where *n* is the length of the string.

Format Specifications

This section describes the format and edit descriptors in detail. In all examples, ƀ represents a blank space.

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* or I*w.m*

O*w*

K*w*

@*w*

Z*w*

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 I*w* 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
ƀ123	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 of type INTEGER*4 or INTEGER*2.

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, B, C, D, E, F, and a blank. 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 accepted. The variable receiving the hexadecimal input must be an INTEGER*4 or INTEGER*2 data type.

The input statement

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

can read any of the following input values:

Input Field	Equivalent Assignment
123456	ihex = 291 (decimal equivalent of 123 hex)
1234	ihex = 291
123	ihex = 291
12	ihex = 288 (decimal equivalent of 120 hex)
1	ihex = 256 (decimal equivalent of 100 hex)
b̸1	ihex = 16 (decimal equivalent of 10 hex)
b̸b̸1	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 I*w.m* form is used and the output value is less than *m* positions, the value is preceded by zeros. If *m* = 0, a 0 value is output as all blanks.

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

```
      PROGRAM int_outputs

      INTEGER*4 int
      int = 12

      WRITE(6, '(11X, "I9", 15X, "O11", 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, "}")

      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}   {   1 152300}   {  1 D4C0}
{  1200000}   {  22 047600}   { 12 4F80}
{ 12000000}   { 267 015400}   { B7 1B00}
{120000000}   {3447 007000}   { 727 0E00}
{*********}   {***********}   {4786 8C00}
```

NOTE

A blank is put between machine word boundaries, so output differs between machines. For example, 352300 octal is output as 1 152300 on 16-bit machines.

Real Format Descriptors: F, D, E, G

The F, D, E, and G format descriptors provide formatting for REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX data types. COMPLEX values are treated as pairs of REAL values. The general specifications are:

F*w*.*d*

E*w*.*d* or E*w*.*d*E*e*

D*w*.*d*

G*w*.*d* or 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 or DOUBLE PRECISION 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
␢123	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
123.E3	a = 123000.0
1234E-3	a = 0.01234
123.E-3	a = 0.123
1234D3	a = 0.1234D+05
123.D3	a = 0.123D+06
blanks	a = 0.0
↑ First column of input field	

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 that 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 E*w*.*d*E*e* format changes the field width allocated for the exponent, which is a field width of two if E*e* 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 except that the letter `D` may be printed ahead of the exponent instead of 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 G*w.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 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 in order 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 this program looks like this:

```
         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}
```

NOTE

The exact output of floating point numbers might vary between systems.

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 six digits of the fraction 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 were negative, the sign would take up another one of these places, so that the minimum negative number would be -9999.999. When a number is out of range for the specified field, the field is filled with asterisks. This illustrates 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:

`A` or `A`*w*

`R` or `R`*w*

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 JaneƀƀƀƀƀƀSmithƀƀƀƀƀ
My name is Jane Smithƀ
```

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.

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

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.

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

```
ABCb̸b̸b̸b̸b̸b̸b̸
```

and the value stored in `char2` is

```
ƛƛƛƛƛƛƛABC
```

where ƛ 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 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:

```
ƀƀƀƀƀABCDE
ƀƀƀƀƀABCDE
```

The differences between the A and R descriptors are shown in Figure 2-1. The upward arrow indicates input and the downward arrow indicates output.

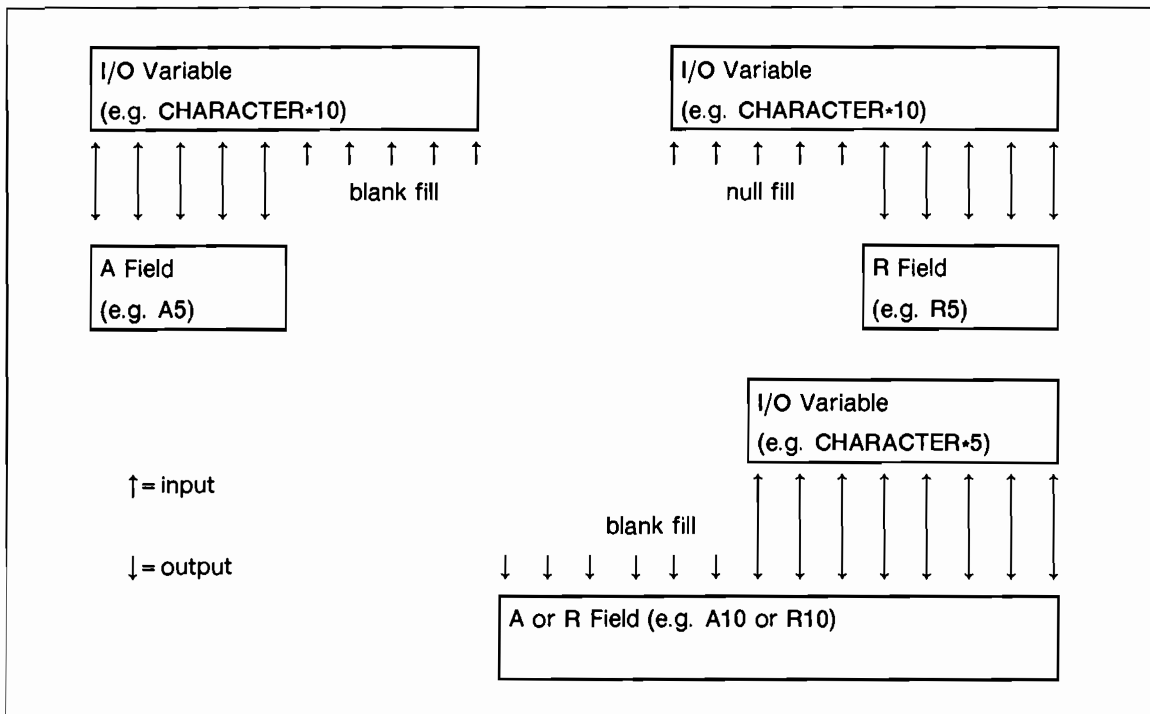


Figure 2-1. Differences Between the A and R Descriptors

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

* INPUT using Aw and Rw format descriptors:

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

*     Input Statement:                  Equivalent assignment:

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

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

* OUTPUT using Aw and Rw format descriptors:

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

*     Output Statement:                 Characters written:

      WRITE(6, '(1X, A6)') alpha3a         !  ^^^abc
      WRITE(6, '(1X, R6)') alpha3a         !  ^^^abc
*                                         (^ represents a blank character)

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

      END
```

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. The lowercase letters `t` and `f` are also allowed. If the first nonblank characters are not `T`, `.T`, `F`, or `.F`, an error occurs.

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

```
b̸b̸b̸Tb̸
F1
```

the value stored in `logical1` is *.TRUE.* and the value stored 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:

```
bbbbF
bT
```

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. A new line is read and 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. Another new line is read, flow control returns to the specification (I5, F5.1), and i2 is input with format I5 and a3 is input with format F5.1.

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.

The NL, NN, and $ edit descriptors work differently on each operating system. For a detailed description, refer to the appendix in the *HP FORTRAN 77 Reference Manual* specific to your operating system.

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

```
bb123bbbbb1231234.5678
```

the program produces this output:

```
bbbbb123bbbbb123bbbbb1234.5676
```

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

```
      bbbbb12.3bbbbb123.45
           ↑        ↑
        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:

```
 123.4bbbbb1234.11
```

Handling Literal Data

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

The ' and " Descriptors

The ' and " edit descriptors write character strings; the paired symbols delimit a string of characters, which can include blanks. Using the ' and " descriptors is preferred over using the H edit descriptor.

If the character string contains a single or double 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 single quotation marks
100   FORMAT (1X, 'Enter your name:')

      WRITE (6,200)      ! String uses double 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

*n*H*string*

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 compatability 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

*n*P

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 *n*th 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 G*w*.*d* is interpreted as E*w*.*d*.

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

*n*PD*w*.*d*

*n*PE*w*.*d* or *n*PE*w*.*d*E*e*

*n*PG*w*.*d* or *n*PG*w*.*d*E*e*

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 *n*P 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:

*n*PF*w*.*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

Depending upon your system, the exponent might be identified with a D or E.

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 commas 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

The exact output of floating point numbers might vary between systems.

Note that the P descriptor has no affect 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) descriptor conditionally terminates format control, just as if the final right parenthesis in the FORMAT statement had 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 as follows:

```
12
24
36
48
```

the program

```
      PROGRAM example1

      OPEN(9, FILE='datafile')
      READ (9,10) i
      READ (9,10) j
10    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, and I 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

`1bb4b.b2b1b`

the variables `int`, `val1`, `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

```
1ƀƀƀƀ2ƀƀ3ƀƀƀƀ
```

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, and I 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

```
1ƀƀƀ4ƀ.ƀ2ƀ1ƀ
```

the variables `int`, `val1`, `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

```
1ƀƀƀƀ2ƀƀ3ƀƀƀƀ
```

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 I/O statements. 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

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

*  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. Here is an implied DO loop:

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


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

```
REAL 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 the list, the entire array is transmitted. For example, the statements

```
REAL 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

```
REAL 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 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
```

Chapter 3
File Handling

HP FORTRAN 77 performs input/output operations with a wide range of devices, including disc 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 disc files and internal files.

Disc 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 disc 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 disc 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. The most 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 we 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 we 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 we 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 exists, 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 disc. This is the default file status.
'DELETE'	The file will be purged from the disc.

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 disc.

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 number 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 to `ios` are summarized in Table 3-3.

Table 3-3. The IOSTAT Specifier

VALUE OF ios	MEANING
0	No error occurred
Greater than 0	An error occurred; an error message number is returned

For example, 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 program.

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_specifiers

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

*  Create file "pfile"; if it already exists go to
*  statement 999; if an error occurs, place error number in "ios":
      OPEN(9, FILE='pfile', STATUS='NEW', ERR=999, IOSTAT=ios)

*  Prompt for data and read data from the standard I/O device:
      WRITE(6, '(1X, "Enter number of products ")')
      READ(5, *) number

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

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

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

999   CALL report_error(ios, 'OPEN', 'data_file')
      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 = 940)

      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
```

NOTE

File error numbers differ between machines.

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
`ios`	Contains the I/O error number
`stmt_name`	Contains the statement that caused the error
`file_name`	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 "Files already exists" (reported for `STATUS='NEW'` files) generate positive return values for `ios`. The exact error numbers vary depending on the system you are using. After the error routine, the STOP statement halts the program and prints the message `Error termination`.

A sample session of the program looks like this:

```
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
```

NOTE

Depending on how carriage control is implemented on your system, the cursor might follow the prompt, or might be on the next line.

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

```
 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

*  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)

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

*  Read the file to get the data;
*  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_fount
      CHARACTER     stmt_name*(*), file_name*(*)
      PARAMETER     (eof = -1, found = 918, not_found = 940)

      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
      DATA          number_of_records/0/

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

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

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

*  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

*  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 shown so far have been sequential access files; unless specified otherwise in the OPEN statement, files are opened for sequential access. But, FORTRAN files can be accessed (read or written) in two ways: sequentially or by direct access.

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. The file pointer in a sequential access file does the following:

- 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.

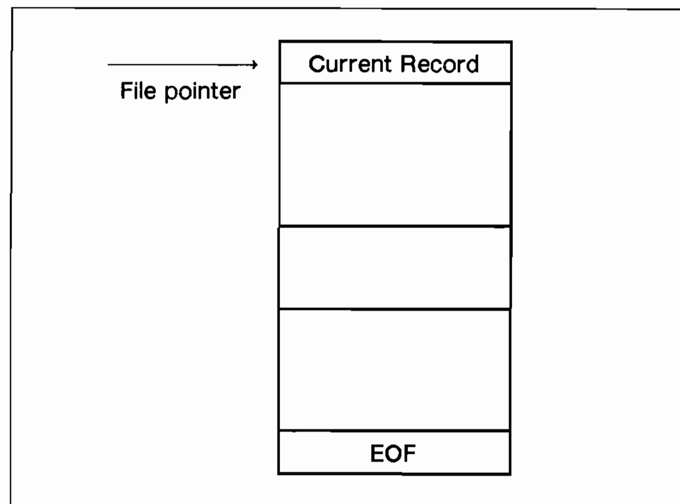


Figure 3-1. Sequential Access Files

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 1. 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.

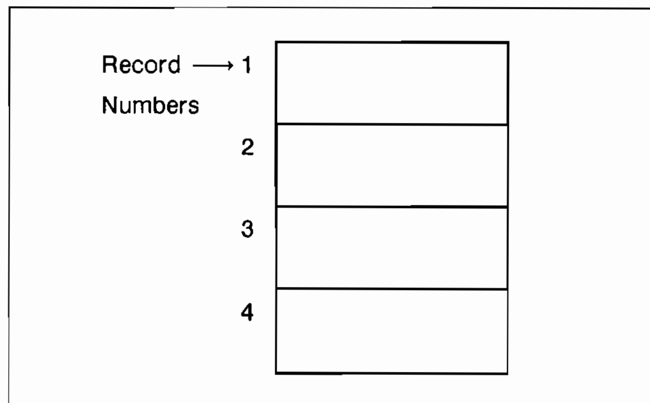


Figure 3-2. Direct Access Files

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'

If FORM='FORMATTED' is specified, BLANK has a default 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, you 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 2 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 4.

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

```
      PROGRAM direct_access

      INTEGER quantity

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

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

*  Prompt for number of transactions:

      WRITE(6,*) "Enter number of transactions:"
      READ(5,*) number

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

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

         WRITE(6,*) "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

*  Open the file "dfile"

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

*  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.

Unformatted I/O

Unformatted input and output 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. The data values can appear in any position and are separated by commas. Unformatted I/O cannot be used for internal 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')

*  Unformatted input statements from unit 9.
*  (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 lines in their internal form. Values are written to the specified output unit without any format conversion. The output list in the WRITE statement must not have more values than can fit into one record.

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 format 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'/

*  Open the formatted and unformatted sequential files:

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

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

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

*  Write the 24 data bytes to the unformatted file
*        (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 disc.

The unformatted WRITE statement writes the data to the file without conversion. The number of bytes written to the disc 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 disc 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 0 if there was no error, or is a system-defined value greater than 0 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. These statements cannot be used on direct access files.

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 statements 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, control transfers to statement 99 and the error number is stored in the variable ios.

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

*  Open the file connected to unit 10:

      OPEN(10, FILE = 'pfile')

*  Read some records:

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

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

      BACKSPACE 10

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

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

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

      ENDFILE 10

      STOP

*  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 disc file `newfile`. 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',
     1   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 terminal
   33 FORMAT (1X, 'The average is ', F12.6)
      CLOSE(3)
      STOP

C  If there is an error in the OPEN or READ, output to a
C  preconnected terminal.
   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 old_file 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 new_file 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 statement, any file positioning statements, or any file status statements with internal files.

Reading from an Internal File

Data is 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_read

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

*  Write into the internal file:

      WRITE(course, '(A10)') a, b, c, d
*  Check the contents of the internal file by reading the
*  file and printing the contents:
      READ(course, '(A10)') a, b, c, d
      WRITE(6,*) a
      WRITE(6,*) b
      WRITE(6,*) c
      WRITE(6,*) d

      END
```

The output of this program looks like this:

```
 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'/

*  Write the character string to the internal file:
      int_file = string

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

*  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
```

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 10, 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/

*  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:

ELEMENT	VALUE
name(1)	14
name(2)	57
name(3)	0
name(4)	-123
name(5)	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/

*  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

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

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

*  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))
```

Chapter 4
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 is a sequence of FORTRAN statements, such as a main program or a subprogram. Table 4-1 summarizes the components of program units.

Table 4-1. 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 a program unit.	Defined in a program unit.
Block data subprogram	Provides initial values for named 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 for 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
```

An example of a program that indirectly calls itself is a subroutine that calls another 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 a Subroutine

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 element 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 a Subroutine

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 which 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 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 *n*
10	1
20	2
30	3

In this example, because **n** is equal to 2, control returns to statement 20. The value of **i** will be set to 2. 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 is shown below. 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 these categories:

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, which then defines 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 associated 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 associated 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 gtfwrd(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: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, and LOGICAL. 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,*) 'Enter a string: '
         READ (5,'(A)') input_string
         result = stringtrim(input_string)
         WRITE (6,'(A)') result
      END DO
      END

      CHARACTER*(*) FUNCTION stringtrim(string)
      CHARACTER*(*) string
      INTEGER 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

      DO 10 i = 1, right - left + 1
         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 ƀ represents a blank, looks like this:

```
      Enter a string: string
      string
      Enter a string: ƀƀƀfour
      four
      Enter a string: threeƀƀƀ
      three
      Enter a string:
```

NOTE

> Depending on how carriage control is implemented on your system, the cursor might follow the prompt, or might be on the next line.

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,
     1               " 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)
```

Here are some more examples of statement 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 or an array 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; 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 variables `i` and `j` and a real function name `f`. The expression `i + j` is converted to real.

Intrinsic Functions

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. Chapter 5 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.

An actual argument can be a variable (simple or subscripted), an array name, a substring, a subprogram name, a constant, an expression, or an alternate return specifier. The actual arguments for statement functions are limited to variables, constants, and expressions. A dummy argument can be a variable or an array name.

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(arry, z, in1, tmp, f)
      DIMENSION arry(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 in the second position of the subroutine argument list (`z`) 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 (not dimensioned) or an array (dimensioned) in the dummy argument list (`tmp`). `fcn` is a function name; `f`, therefore, 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

```
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 double precision variable, and 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. Thus, 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 length is (*), 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*10  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 redefined 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 4, then 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

```
PROGRAM 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    right(24)
.
.
.
CALL mysub(right)
.
.
.
END
```

the subroutine mysub must include a similar declaration, like this:

```
SUBROUTINE mysub(wrong)
INTEGER*2    wrong(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 declarators, 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 n assume the values passed to them through i and j. These variables are used in a 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 **z** can take any value from 1 to 10. The last subscript of the array **arry** 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(arry, k)
INTEGER arry(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 declarators cannot be used in COMMON statements, in main programs, or in BLOCK DATA subprograms.

Passing 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 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 this 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 COMMON in an INCLUDE file.

Variables in blank common blocks can never 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 subprogram. The subroutine uses the values inputted 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. The statement looks like this:

```
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 used 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 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 labeled common blocks. Block data subprograms can also initialize the variables and arrays declared in the common block. A block data subprogram is the only compilation unit in which variables and arrays in a named common block can be initialized. 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. Block2 contains i and the 10 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 named common block referenced in an executable FORTRAN program can be defined in a block data subprogram.

In general, only specification statements and data initialization statements are allowed in the body of a block data subprogram. Block data statements cannot contain executable statements. Acceptable statements include: COMMON, DIMENSION, type statements (INTEGER*4, REAL*8, etc.), DATA, SAVE, PARAMETER, and IMPLICIT statements. EXTERNAL and INTRINSIC statements are not allowed. Blank common variables cannot be initialized in a block data subprogram.

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 (not labeled 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**.

Calling Subprograms Written in Other Languages

Subroutine and function subprograms can be written in languages other than FORTRAN. Subprogram names and dummy arguments must conform to FORTRAN requirements. See the system-specific appendix in this manual for details on data restrictions.

For example, the FORTRAN program

```
PROGRAM otherlang
INTEGER num
num = 5
CALL sub(num)
PRINT *, 'num = ', num
END
```

can call this Pascal subroutine:

```
$SUBPROGRAM$

PROGRAM pasex;

PROCEDURE sub
   (VAR value: INTEGER);

BEGIN
     value := value * value;
END;

BEGIN
END.
```

The FORTRAN program and the Pascal subroutine are compiled separately. The subroutine is then linked to the main program to produce this output:

```
num = 25
```

For details on compiling and linking, refer to the system-specific appendix in the *HP FORTRAN 77 Reference Manual*.

Chapter 5
Intrinsic Functions

An intrinsic function is a built-in function that returns a single value or is a subroutine that performs a particular task. Intrinsic functions convert values from one data type to another, perform data manipulation, and also perform basic mathematical functions, such as calculating sines, cosines, and square roots of numbers.

This chapter describes the intrinsic functions, including the generic and specific names of the functions, the number of required arguments, the data types of arguments allowed, and the data type of the returned value.

Invoking an Intrinsic Function

An intrinsic function is invoked when the function name and any arguments appear in an expression.

For example, the statement

```
squareroot = SQRT (value1 + value2)
```

computes the square root of `value1 + value2`.

Some intrinsics are actually subroutines. For example, the intrinsic MVBITS is a subroutine name.

You can define and call your own function subprogram with the same name as an intrinsic function. However, the intrinsic function will be used unless your function is declared as an external function with the EXTERNAL statement. Defining your own function subprogram is described in Chapter 4.

Declaring an intrinsic function in a type statement has no effect on the type of the intrinsic function. For example, the statements

```
INTEGER float
x = float(y)
```

do not change the data type of `float` to INTEGER; the type of `float` remains REAL. Also, an IMPLICIT statement does not change the type of an intrinsic function.

Generic and Specific Function Names

Each of the intrinsic functions have a generic name and/or one or more specific names. Either the generic or the specific name can call the function; however, generic names are recommended.

Generic names are used with any of the valid data types for the particular function. Generic names simplify calling the intrinsic functions because the same name can be used with more than one type of argument.

For example, in the generic function `COS(arg)`, `arg` can be a real, double precision, complex, or double complex value. The data type of the result will be the same data type as the argument.

Specific names are used with a particular data type. For example, in the specific function `DCOS(arg)`, `arg` is a double precision value. The data type of the result is double precision. Similarly, in the specific function `CCOS(arg)`, `arg` is a COMPLEX*8 value. The data type of the result is COMPLEX*8.

If a function, such as MOD, requires more than one argument, all arguments in that function must be the same data type. That is, short integer (INTEGER*2) arguments can be mixed with integer arguments (INTEGER*4), but integer and real arguments cannot be mixed.

If a generic or specific name appears as a formal argument, that name does not identify an intrinsic function in that program unit or statement function. For example, in this subroutine

```
SUBROUTINE x(log, f)
.
.
.
f = log(f)

END
```

`log` is not an intrinsic function.

Summary of the Intrinsic Functions

The intrinsic functions are summarized in Tables 5-1 through 5-5. The functions are categorized as follows: Table 5-1 contains the arithmetic functions, Table 5-2 contains the bit manipulation functions, Table 5-3 contains the character functions, Table 5-4 contains the numeric conversion functions, and Table 5-5 contains the transcendental functions.

The following terms are applicable to Tables 5-1 through 5-5 and the function descriptions:

- Real is REAL*4
- Double is REAL*8
- Integer is INTEGER*2 or INTEGER*4
- Logical is LOGICAL*2 or LOGICAL*4
- Complex is COMPLEX*8 or COMPLEX*16

Table 5-1. Arithmetic Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute value	$\lvert a \rvert$	1	ABS	IABS	Integer*4	Integer*4
				HABS	Integer*2	Integer*2
				ABS	Real	Real
				DABS	Double	Double
				CABS	Complex*8	Real
				ZABS†	Complex*16	Double
Remaindering	a−INT(a/b)*b	2	MOD	MOD	Integer*4	Integer*4
				HMOD	Integer*2	Integer*2
				AMOD	Real	Real
				DMOD	Double	Double
Transfer of sign	$\lvert a \rvert$ if $b \geq 0$ $-\lvert a \rvert$ if $b < 0$	2	SIGN	ISIGN	Integer*4	Integer*4
				HSIGN†	Integer*2	Integer*2
				SIGN	Real	Real
				DSIGN	Double	Double
Positive difference	$a-b$ if $a > b$ 0 if $a \leq b$	2	DIM	IDIM	Integer*4	Integer*4
				HDIM†	Integer*2	Integer*2
				DIM	Real	Real
				DDIM	Double	Double
Double precision product	a*b	2		DPROD	Real	Double
Choosing largest value	max(a,b,...)	≥2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
				AMAX0	Integer	Real
				MAX1	Real	Integer
Choosing smallest value	min(a,b,...)	≥2	MIN	MIN0	Integer	Integer
				AMIN1	Real	Real
				DMIN1	Double	Double
				AMIN0	Integer	Real
				MIN1	Real	Integer
Imaginary part of a complex argument	ai	1	IMAG	AIMAG	Complex*8	Real
				DIMAG†	Complex*16	Double
Conjugate of a complex argument	(ar,-ai)	1	CONJG	CONJG	Complex*8	Complex*8
				DCONJG†	Complex*16	Complex*16
Logical product		2	IAND	IAND†	Integer*4	Integer*4
				HIAND†	Integer*2	Integer*2
Logical sum		2	IOR	IOR†	Integer*4	Integer*4
				HIOR†	Integer*2	Integer*2
Exclusive OR		2	IXOR IEOR	IEOR†	Integer*4	Integer*4
				HIEOR†	Integer*2	Integer*2
Complement		1	NOT	NOT†	Integer*4	Integer*4
				HNOT†	Integer*2	Integer*2

† Indicates the function is an extension to the ANSI 77 standard.

Table 5-2. Bit Manipulation Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Bit test		2	BTEST	BTEST†	Integer*4	Logical
				HTEST†	Integer*2	Logical
Bit set		2	IBSET	IBSET†	Integer*4	Integer*4
				HBSET†	Integer*2	Integer*2
Bit clear		2	IBCLR	IBCLR†	Integer*4	Integer*4
				HBCLR†	Integer*2	Integer*2
Bit move		5	MVBITS	MVBITS†	Integer*4	Integer*4
				HMVBITS†	Integer*2	Integer*2
Logical shift		2	ISHFT	ISHFT†	Integer*4	Integer*4
				HSHFT†	Integer*2	Integer*2
Circular shift		3	ISHFTC	ISHFTC†	Integer*4	Integer*4
				HSHFTC†	Integer*2	Integer*2
Bit extraction		3	IBITS	IBITS†	Integer*4	Integer*4
				HBITS†	Integer*2	Integer*2

† Indicates the function is an extension to the ANSI 77 standard.

Table 5-3. Character Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Conversion to character		1		CHAR	Integer	Character
Length	Length of character entry	1		LEN	Character	Integer
Index of a substring	Location of substring b in string a	2		INDEX	Character	Integer
Lexically greater than or equal	a ≥b	2		LGE	Character	Logical
Lexically greater than	a >b	2		LGT	Character	Logical
Lexically less than or equal	a ≤b	2		LLE	Character	Logical
Lexically less than	a <b	2		LLT	Character	Logical

Table 5-4. Numeric Conversion Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to integer	1	INT	–	Integer	Integer
				INT	Real	Integer
				IFIX	Real	Integer
				IDINT	Double	Integer
				–	Complex	Integer
	Conversion to real	1	REAL	FLOAT	Integer	Real
				–	Real	Real
				SNGL	Double	Real
				–	Complex	Real
	Conversion to double precision	1	DBLE	DFLOAT†	Integer	Double
				–	Real	Double
				–	Double	Double
				–	Complex	Double
	Conversion to complex*8	1 or 2	CMPLX	–	Integer	Complex*8
				–	Real	Complex*8
				–	Double	Complex*8
				–	Complex‡	Complex*8
	Conversion to complex*16	1 or 2	DCMPLX†	–	Integer	Complex*16
				–	Real	Complex*16
				–	Double	Complex*16
				–	Complex‡	Complex*16
	Conversion to integer	1		ICHAR	Character	Integer
	Conversion to character	1		CHAR	Integer	Character
Truncation	REAL(INT(a)) or DBLE(INT(a))	1	AINT	AINT	Real	Real
				DINT	Double	Double
				DDINT†	Double	Double
Nearest whole number	INT(a+.5) if a ≥0 INT(a–.5) if a <0	1	ANINT	ANINT	Real	Real
				DNINT	Double	Double
Nearest integer	INT(a+.5) if a ≥0) INT(a–.5) if a <0	1	NINT	NINT	Real	Integer
				IDNINT	Double	Integer

† Indicates the function is an extension to the ANSI 77 standard.

‡ If type COMPLEX is used, there can be only one argument.

Table 5-5. Transcendental Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Sine	sin(a)	1	SIN	SIN	Real	Real
				DSIN	Double	Double
				CSIN	Complex*8	Complex*8
				ZSIN†	Complex*16	Complex*16
Cosine	cos(a)	1	COS	COS	Real	Real
				DCOS	Double	Double
				CCOS	Complex*8	Complex*8
				ZCOS†	Complex*16	Complex*16
Tangent	tan(a)	1	TAN	TAN	Real	Real
				DTAN	Double	Double
				CTAN†	Complex*8	Complex*8
				ZTAN†	Complex*16	Complex*16
Arcsine	arcsin(a)	1	ASIN	ASIN	Real	Real
				DASIN	Double	Double
Arccosine	arccos(a)	1	ACOS	ACOS	Real	Real
				DACOS	Double	Double
Arctangent	arctan(a)	1	ATAN	ATAN	Real	Real
				DATAN	Double	Double
	arctan(a/b)	2	ATAN2	ATAN2	Real	Real
				DATAN2	Double	Double
Hyperbolic sine	sinh(a)	1	SINH	SINH	Real	Real
				DSINH	Double	Double
Hyperbolic cosine	cosh(a)	1	COSH	COSH	Real	Real
				DCOSH	Double	Double
Hyperbolic tangent	tanh(a)	1	TANH	TANH	Real	Real
				DTANH	Double	Double
Hyperbolic arcsine	asinh(a)	1	ASINH	ASINH†	Real	Real
				DASINH†	Double	Double
Hyperbolic arccosine	acosh(a)	1	ACOSH	ACOSH†	Real	Real
				DACOSH†	Double	Double
Hyperbolic arctangent	atanh(a)	1	ATANH	ATANH†	Real	Real
				DATANH†	Double	Double
Square root	a**(1/2)	1	SQRT	SQRT	Real	Real
				DSQRT	Double	Double
				CSQRT	Complex*8	Complex*8
				ZSQRT†	Complex*16	Complex*16
Exponential	e**a	1	EXP	EXP	Real	Real
				DEXP	Double	Double
				CEXP	Complex*8	Complex*8
				ZEXP†	Complex*16	Complex*16

† Indicates the function is an extension to the ANSI 77 standard.

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Natural logarithm	log(a)	1	LOG	ALOG	Real	Real
				DLOG	Double	Double
				CLOG	Complex*8	Complex*8
				ZLOG†	Complex*16	Complex*16
Common logarithm	log10(a)	1	LOG10	ALOG10	Real	Real
				DLOG10	Double	Double

† Indicates the function is an extension to the ANSI 77 standard.

Function Descriptions

The generic functions, listed in alphabetical order, are described below. The functions that do not have a generic name are listed alphabetically by specific name.

NOTE

The floating point value returned to the function might vary between systems.

ABS

`ABS(arg)` is a generic function that returns the absolute value of an integer, real, complex, or double precision argument. A complex value is expressed as an ordered pair of real or double precision numbers in the form `(ar, ai)`, where `ar` is the real part and `ai` is the imaginary part. If `arg` is complex, `ABS(arg)` is equal to the square root of `(ar**2 + ai**2)`.

For integer, real, and double precision arguments, the result is the same data type as the argument. For complex arguments, the result is real.

Examples:

Function Call	Value Returned to *a*
`a = ABS(100)`	100
`a = ABS(-100.0)`	100.0
`a = ABS(vector)`	28.32716, where `vector` = (12.84, 25.25)
`a = ABS(-1.23451234512345D2)`	123.451234512345

The specific function names are ABS for real arguments, CABS for COMPLEX*8 arguments, DABS for double precision arguments, IABS for INTEGER*4 arguments, HABS for INTEGER*2 arguments, and ZABS for COMPLEX*16 arguments.

ACOS

`ACOS(arg)` is a generic function that returns the arccosine of a real or double precision argument. The value of |`arg`| must be less than or equal to one. The result is expressed in radians and is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = ACOS(0.0628)`	1.5079550
`a = ACOS(0.0)`	1.5707964
`a = ACOS(0.0D0)`	1.570796326794897

The specific function names are ACOS for real arguments and DACOS for double precision arguments.

ACOSH

`ACOSH(arg)` is a generic function that returns the hyperbolic arccosine of a real or double precision argument. The argument must be greater than or equal to one and less than or equal to the maximum number allowed on your system. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = ACOSH(1.2)`	0.622362
`a = ACOSH(1.0)`	0.0
`a = ACOSH(1.0D0)`	0.0

The specific function names are ACOSH for real arguments and DACOSH for double precision arguments.

AINT

AINT(arg) is a generic function that truncates the fractional digits from a real or double precision argument. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = AINT(324.7892)	324.0
a = AINT(324.7892D2)	32478.0

The specific function names are AINT for real arguments and DINT and DDINT for double precision arguments. If |arg| is less than one, the result is zero. If |arg| is greater than one, the result is the value with the same sign as arg with a magnitude that does not exceed |arg|.

ANINT

ANINT(arg) is a generic function that returns the nearest whole number. The argument can be real or double precision. The result is INT (arg + 0.5) if arg is positive or zero, and is INT (arg - 0.5) if arg is negative. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = ANINT(-678.44)	-678.0
a = ANINT(678.44)	678.0
a = ANINT(0.00)	0.0
a = ANINT(6.78 D1)	68.0

The specific function names are ANINT for real arguments and DNINT for double precision arguments.

ASIN

ASIN(**arg**) is a generic function that returns the arcsine of a real or double precision argument. The value of |**arg**| must be less than or equal to one. The result is expressed in radians and is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = ASIN(0.30)	0.304692
a = ASIN(0.30D0)	0.3046926540153975

The specific function names are ASIN for real arguments and DASIN for double precision arguments.

ASINH

ASINH(**arg**) is a generic function that returns the hyperbolic arcsine of a real or double precision argument; the result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = ASINH(0.30)	0.2956731
a = ASINH(0.30D0)	0.295673047563423

The specific function names are ASINH for real arguments and DASINH for double precision arguments.

ATAN

ATAN(arg) is a generic function that returns the arctangent of a real or double precision argument. The result is expressed in radians and is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = ATAN(1.0)	0.7853982
a = ATAN(3.141592653D0)	1.262627255624651

The specific function names are ATAN for real arguments and DATAN for double precision arguments.

ATANH

ATANH(arg) is a generic function that returns the hyperbolic arctangent of a real or double precision argument. The value of |arg| must be less than one. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = ATANH(0.30)	0.3095196
a = ATANH(0.30D0)	0.309519604203112

The specific function names are ATANH for real arguments and DATANH for double precision arguments.

ATAN2

`ATAN2(arg1, arg2)` is a generic function that returns the arctangent of **arg1/arg2**. The arguments can be real or double precision. The result is expressed in radians and is the same data type as the arguments. The arguments cannot both be zero.

Examples:

Function Call	Value Returned to *a*
`a = ATAN2(1.0, 2.0)`	0.4636476
`a = ATAN2(3.141592653D0, 1.0D0)`	1.262627255624651

The specific function names are ATAN2 for real arguments and DATAN2 for double precision arguments.

BTEST

`BTEST(arg1, arg2)` is a generic function that tests individual bits of storage. The arguments are integer and the result is logical. If the *arg2*th bit of **arg1** is equal to 1, the result is true. If the *arg2*th bit is equal to 0, the result is false. If **arg2** is greater than or equal to the bit size of **arg1**, the result is false. Bit positions are numbered right to left, with the rightmost bit numbered 0.

Examples:

Function Call	Value Returned to `l`
`l = BTEST(3, 0)`	T
`l = BTEST(0, 0)`	F
`l = BTEST(0, 3)`	F
`l = BTEST(4, 1)`	F

The specific function names are BTEST for INTEGER*4 arguments and HTEST for INTEGER*2 arguments.

CHAR

`CHAR(i)` is a specific function that returns the character value in the *i*th position of the ASCII collating sequence. The argument is integer and the result is character.

Examples:

Function Call	Value Returned to c
c = CHAR(97)	'a'
c = CHAR(122)	'z'
c = CHAR(53)	'5'

There is no generic name for this function.

CMPLX

`CMPLX(arg)` or `CMPLX(arg1, arg2)` is a specific function that performs type conversion to a complex value. CMPLX can have one or two arguments. If you specify one argument, the argument can be integer, real, double precision, or complex. If you specify two arguments, the arguments must be the same type and both must be integer, real, or double precision.

For one argument not of type COMPLEX, the result is complex, with `REAL(arg)` used as the real part and the imaginary part equal to zero. For one argument of type COMPLEX, the result is the same as the argument, or as much of the argument that can fit in a COMPLEX*8 variable. For two arguments, `arg1` and `arg2`, the result is complex, with `REAL(arg1)` used as the real part and `REAL(arg2)` used as the imaginary part.

Examples:

Function Call	Value Returned to *a*
c = CMPLX(1.0)	(1.0, 0.0)
c = CMPLX(1.0, 1.0)	(1.0, 1.0)
c = CMPLX(1,0)	(1.0, 0.0)
c = CMPLX(3.141592653D0, 0.0D0)	(3.1415927, 0.0)

There is no generic name for this function.

CONJG

`CONJG(arg)` is a generic function that returns the conjugate of a COMPLEX*8 or COMPLEX*16 argument. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = CONJG(var1)`	(3.0, 0.0), where `var1` = (3.0, 0.0)
`a = CONJG(var2)`	(3.0, -1.0), where `var2` = (3.0, 1.0)

The specific function names are CONJG for COMPLEX*8 arguments and DCONJG for COMPLEX*16 arguments.

COS

`COS(arg)` is a generic function that returns the cosine of a real, double precision, or complex argument. The argument is expressed in radians. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = COS(0.0)`	1.0
`a = COS(0.0628)`	0.9980288

The specific function names are COS for real arguments, CCOS for COMPLEX*8 arguments, DCOS for double precision arguments, and ZCOS for COMPLEX*16 arguments.

COSH

COSH(arg) is a generic function that returns the hyperbolic cosine of a real or double precision argument. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = COSH(1.0)	1.5430807
a = COSH(3.0)	10.06766
a = COSH(3.0D0)	10.0676619957778

The specific function names are COSH for real arguments and DCOSH for double precision arguments.

DBLE

DBLE(arg) is a generic function that converts the argument to double precision. The argument can be integer, real, double precision, or complex. For an integer or real argument, the result is as much precision of the significant part of the argument as the argument can provide. For a double precision argument, the result is the argument. For a complex argument, the result is as much precision of the significant real part of the argument as the argument can provide.

Examples:

Function Call	Value Returned to *a*
a = DBLE(4)	4.0
a = DBLE(4.0)	4.0
a = DBLE(4.0D2)	400.0
a = DBLE(var1)	4.0, where var1 = (4.00, 2)

The specific function name is DFLOAT for integer arguments.

DCMPLX

`DCMPLX(arg)` or `DCMPLX(arg1, arg2)` are specific functions that perform type conversion to a double complex value. DCMPLX can have one or two double precision arguments. If you specify one argument, the argument can be integer, real, double precision, or complex. If you specify two arguments, the arguments must be of the same type and both must be integer, real, or double precision.

For one argument not of type COMPLEX, the result is COMPLEX*16, with `DBLE(arg)` used as the real part and the imaginary part equal to zero. For one argument of type COMPLEX, the result is the same as the argument. For two arguments, `arg1` and `arg2`, the result is COMPLEX*16, with `DBLE(arg1)` used as the real part and `DBLE(arg2)` used as the imaginary part.

Examples:

Function Call	Value Returned to *c*
`c = DCMPLX(1.0)`	(1.0, 0.0)
`c = DCMPLX(1.0, 0.0)`	(1.0, 0.0)
`c = DCMPLX(3.141592653D0, 0.0D0)`	(3.1415927, 0.0)

There is no generic name for this function.

DIM

`DIM(arg1, arg2)` is a generic function that returns a positive difference. The arguments must be the same data type and can be integer, real, or double precision. The result is the same type as the arguments. The result is `arg1 - arg2` if `arg1` is greater than `arg2`. The result is zero if `arg1` is less than or equal to `arg2`.

Examples:

Function Call	Value Returned to *a*
`a = DIM(56.9, 45.4)`	11.5
`a = DIM(45.4, 56.9)`	0.0
`a = DIM(45.4, 45.4)`	0.0

The specific function names are DIM for real arguments, DDIM for double precision arguments, IDIM for INTEGER*4 arguments, and HDIM for INTEGER*2 arguments.

DPROD

DPROD(arg1, arg2) is a specific function that returns the double precision product of two real arguments (arg1 * arg2). The result is a REAL*8 number with none of the fractional portion lost and is equal to DBLE(arg1) * DBLE(arg2).

Examples:

Function Call	Value Returned to *d*
d = DPROD(2.2, 2.2)	4.840
d = DPROD(1.0, 2.0)	2.0

There is no generic name for this function.

EXP

EXP(arg) is a generic function that returns an exponential result (e**arg). The argument can be real, double precision, or complex. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = EXP(3.0)	20.08554
a = EXP(1.5D1)	3269017.37247211
a = EXP(var)	(10.85226, 16.90140), where var = (3.0, 1.0)

The specific function names are EXP for real arguments, DEXP for double precision arguments, CEXP for COMPLEX*8 arguments, and ZEXP for COMPLEX*16 arguments.

IAND

IAND(arg1, arg2) is a generic function that returns the logical product, or bitwise AND, of two integer arguments. The result is integer.

Examples:

Function Call	Value Returned to *i*
i = IAND(0, 0)	0
i = IAND(1, 0)	0
i = IAND(0, 1)	0
i = IAND(1, 1)	1

The specific function names are IAND for INTEGER*4 arguments and HIAND for INTEGER*2 arguments.

IBCLR

IBCLR(arg1, arg2) is a generic function that returns arg1 with the *arg2*th bit cleared (set to 0). If arg2 is greater than or equal to the bit size of arg1, the result is equal to arg1. The arguments and result are integer.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

Examples:

Function Call	Value Returned to *i*
i = IBCLR(3, 4)	3
i = IBCLR(1, 2)	1
i = IBCLR(1, 0)	0

The specific function names are IBCLR for INTEGER*4 arguments and HBCLR for INTEGER*2 arguments.

IBITS

IBITS(arg1, arg2, arg3) is a generic function that extracts a subfield of **arg3** bits in length from **arg1**, starting with bit position **arg2** and extending left **arg3** bits. The arguments and result are integer. The extracted bits are right-justified in the result with the remaining bits set to 0. The value of **arg2 + arg3** must be less than or equal to 16 if **arg1** is INTEGER*2, or 32 if **arg1** is INTEGER*4.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

Examples:

Function Call	Value Returned to *i*
i = IBITS(3, 4, 8)	0
i = IBITS(16, 4, 8)	1
i = IBITS(12, 2, 2)	3

The specific function names are IBITS for INTEGER*4 arguments and HBITS for INTEGER*2 arguments.

IBSET

IBSET(arg1, arg2) is a generic function that returns the value of **arg1** with the *arg2*th bit set to 1. If **arg2** is greater than or equal to the bit size of **arg1**, the result is **arg1**. The arguments and the result are integer.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

Examples:

Function Call	Value Returned to *i*
i = IBSET(3, 4)	19
i = IBSET(1, 2)	5
i = IBSET(1, 0)	1

The specific function names are IBSET for INTEGER*4 arguments and HBSET for INTEGER*2 arguments.

ICHAR

ICHAR(arg) is a specific function that converts a character argument to an integer value. The result depends on the collating position of the argument in the ASCII collating sequence. If **arg** is longer than one character, the first character is used.

Examples:

Function Call	Value Returned to *i*
i = ICHAR('a')	97
i = ICHAR('Z')	90
i = ICHAR('5')	53

There is no generic name for this function.

IEOR

IEOR(**arg1**, **arg2**) is a generic function that returns the bitwise exclusive OR of two integer arguments. The result is integer.

Examples:

Function Call	Value Returned to *i*
i = IEOR(1, 0)	1
i = IEOR(1, 1)	0
i = IEOR(0, 0)	0

An alternate generic function name is IXOR. The specific function names are IEOR for INTEGER*4 arguments and HIEOR for INTEGER*2 arguments.

IMAG

IMAG(arg) is a generic function that returns the imaginary part of a complex number. The argument can be COMPLEX*8 or COMPLEX*16. For COMPLEX*8 arguments, the result is real; for COMPLEX*16 arguments, the result is double precision. A complex number is expressed as an ordered pair of real or double precision numbers in the form (ar, ai), where ar is the real part and ai is the imaginary part. The result is the real value of ai.

Examples:

Function Call	Value Returned to *a*
a = IMAG(var1)	0.00, where var1 = (25.058, 0.0)
a = IMAG(var2)	3.5, where var2 = (25.3, 3.5)
a = IMAG(var3)	3.5, where var3 = (25.3D0, 3.5D0)

The specific function names are AIMAG for COMPLEX*8 arguments, and DIMAG for COMPLEX*16 arguments.

INDEX

INDEX(arg1, arg2) is a specific function that returns the location of substring arg2 within string arg1. Both arguments must be character strings. If string arg2 occurs as a substring within string arg1, the result is an integer indicating the starting position of the substring arg2 within arg1. The character positions are numbered from left to right with the leftmost character numbered 1. If arg2 does not occur as a substring, the result is zero. If arg2 occurs more than once within arg1, the result is the starting position of the first occurrence. If the length of arg1 is less than the length of arg2, the result is zero.

Examples:

Function Call	Value Returned to *i*
i = INDEX('ABCD', 'BC')	2
i = INDEX('10552', '5')	3
i = INDEX('ABC', 'XY')	0
i = INDEX('ABC', 'abc')	0

There is no generic name for this function.

INT

INT(arg) is a generic function that converts data types to integer. The argument can be integer, real, double precision, or complex; the result is integer. If **arg** exceeds the largest integer allowed, the result will be undefined.

If **arg** is an integer, INT (**arg**) = **arg**. If **arg** is real or double precision and |**arg**| is less than one, the result is zero. If |**arg**| is greater than one, the result of INT (**arg**) is the integer with the same sign as **arg** whose magnitude does not exceed |**arg**|. If **arg** is complex, the real part of **arg** is used and the result is found by applying the rules to the real part of **arg**.

Examples:

Function Call	Value Returned to *i*
i = INT(-3.7)	-3
i = INT(25)	25
i = INT(25.9D0)	25
i = INT(var)	30, where var = (30.57, 0.0)

The specific function names are INT for real arguments, IFIX for real arguments, and IDINT for double precision arguments.

IOR

IOR(arg1, arg2) is a generic function that returns the logical (bitwise) sum (Boolean OR) of two integer arguments. The result is integer.

Examples:

Function Call	Value Returned to *i*
i = IOR(1, 1)	1
i = IOR(0, 0)	0
i = IOR(1, 0)	1

The specific function names are IOR for INTEGER*4 arguments and HIOR for INTEGER*2 arguments.

ISHFT

ISHFT(arg1, arg2) is a generic function that returns the value of **arg1** shifted by **arg2** bit positions. If **arg2** is greater than zero, the shift is to the left; if **arg2** is less than zero, the shift is to the right; if **arg2** equals zero, no shift occurs.

If **arg1** is an INTEGER*2 argument and **arg2** is greater than 15 or **arg2** is less than -15, the result is zero. If **arg2** is an INTEGER*4 and **arg2** is greater than 31 or **arg2** is less than -31, the result is zero.

Bits shifted out from the left or right end are lost. Zeros are shifted in from the opposite end. The result is the same type as the arguments.

Examples:

Function Call	Value Returned to *a*
a = ISHFT(3, 4)	48
a = ISHFT(1, 4)	16
a = ISHFT(1, -4)	0

The specific function names are ISHFT for INTEGER*4 arguments and HSHFT for INTEGER*2 arguments.

ISHFTC

ISHFTC(arg1, arg2, arg3) is a generic function that returns the circular shift of an integer argument. The result is the rightmost **arg3** bits of **arg1** shifted circularly **arg2** places. That is, the bits shifted out of one end are shifted into the opposite end. No bits are lost.

The unshifted bits of the result are the same as the unshifted bits of the argument **arg1**. The absolute value of the argument **arg2** must be less than or equal to **arg3**. The argument **arg3** must be greater than or equal to one and less than or equal to 16 if **arg1** is INTEGER*2, or must be less than or equal to 32 if **arg1** is INTEGER*4. If **arg3** does not fall within this range, the results can be undefined.

Examples:

Function Call	Value Returned to *i*
i = ISHFTC(3, 4, 8)	48
i = ISHFTC(1, 4, 8)	16

The specific function names are ISHFTC for INTEGER*4 arguments and HSHFTC for INTEGER*2 arguments.

IXOR

IXOR(arg1, arg2) is a generic function that returns the bitwise exclusive OR of two integer arguments. The result is integer.

Examples:

Function Call	Value Returned to *i*
i = IXOR(1, 0)	1
i = IXOR(1, 1)	0
i = IXOR(0, 0)	0

The alternate generic function name is IEOR. The specific function names are IEOR for INTEGER*4 arguments and HIEOR for INTEGER*2 arguments.

LEN

LEN(arg) is a specific function that returns the length of a character string. The argument is type character and the result is an integer indicating the length of the argument.

Examples:

Function Call	Value Returned to *i*
i = LEN('string')	6
i = LEN('howlongami')	10

There is no generic name for this function.

LGE

LGE(arg1, arg2) is a specific function that returns a logical result indicating whether arg1 is "Lexically Greater than or Equal to" arg2. The arguments are character strings. The result is true if arg1 is equal to arg2 or if arg1 follows arg2 in the ASCII collating sequence. In all other cases, the result is false. If arg1 and arg2 have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples:

Function Call	Value Returned to 1
1 = LGE('ABC', 'BC')	F
1 = LGE('BC', 'ABC')	T
1 = LGE('ABC', 'ABC')	T

There is no generic name for this function.

LGT

LGT(arg1, arg2) is a specific function that returns a logical result indicating whether arg1 is "Lexically Greater Than" arg2. The arguments are character strings. The result is true if arg1 follows arg2 in the ASCII collating sequence. In all other cases, the result is false. If arg1 and arg2 have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples:

Function Call	Value Returned to 1
1 = LGT('ABC', 'BC')	F
1 = LGT('BC', 'ABC')	T
1 = LGT('ABC', 'ABC')	F

There is no generic name for this function.

LLE

LLE(arg1, arg2) is a specific function that returns a logical result indicating whether arg1 is "Lexically Less than or Equal to" arg2. The arguments are character strings. The result is true if arg1 is equal to arg2 or if arg1 precedes arg2 in the ASCII collating sequence. In all other cases, the result is false. If arg1 and arg2 have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples:

Function Call	Value Returned to l
l = LLE('ABC', 'BC')	T
l = LLE('BC', 'ABC')	F
l = LLE('ABC', 'ABC')	T

There is no generic name for this function.

LLT

LLT(arg1, arg2) is a specific function that returns a logical result indicating whether arg1 is "Lexically Less Than" arg2. The arguments are character strings. The result is true if arg1 precedes arg2 in the ASCII collating sequence. In all other cases, the result is false. If arg1 and arg2 have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples:

Function Call	Value Returned to l
l = LLT('ABC', 'BC')	T
l = LLT('BC', 'ABC')	F
l = LLT('ABC', 'ABC')	F

There is no generic name for this function.

LOG

`LOG(arg)` is a generic function that returns the natural logarithm (logarithm base e) of a real, double precision, COMPLEX*8, or COMPLEX*16 argument; the argument must be greater than zero for real and double precision arguments. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = LOG(6.0)`	1.7917595
`a = LOG(6.0D0)`	1.791759469228055
`a = LOG(var1)`	(1.7917595, 0.00), where var1 = (6.0D0, 0.0D0)

The specific function names are ALOG for real arguments, CLOG for COMPLEX*8 arguments, DLOG for double precision arguments, and ZLOG for COMPLEX*16 arguments.

LOG10

`LOG10(arg)` is a generic function that returns the common logarithm (logarithm base 10) of a real or double precision argument; the argument must be greater than zero. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = LOG10(6.0)`	0.7781513
`a = LOG10(6.0D0)`	0.778151250383644

The specific function names are ALOG10 for real arguments and DLOG10 for double precision arguments.

MAX

`MAX(arg1, arg2, ...)` is a generic function that returns the largest value from the list of arguments. The arguments can be integer, real, or double precision; the number of arguments can vary, but there must be at least two. The result is the same data type as the arguments.

Examples:

Function Call	Value Returned to *a*
`a = MAX(5, -2, 54, 11, 52)`	54
`a = MAX(5.0, 43.24, 44.1, 78.2)`	78.2

The specific function names are AMAX0 for integer arguments with a real result, AMAX1 for real arguments, DMAX1 for double precision arguments, MAX0 for integer arguments, and MAX1 for real arguments with an integer result.

MIN

`MIN(arg1, arg2, ...)` is a generic function that returns the smallest value from the list of arguments. The arguments can be integer, real, or double precision; the number of arguments can vary, but there must be at least two. The result is the same data type as the arguments.

Examples:

Function Call	Value Returned to *a*
`a = MIN(5, -2, 54, 11, 52)`	-2
`a = MIN(5.0, 43.24, 44.1, 78.2)`	5.0

The specific function names are AMIN0 for integer arguments with a real result, AMIN1 for real arguments, DMIN1 for double precision arguments, MIN0 for integer arguments, and MIN1 for real arguments with an integer result.

MOD

MOD(arg1, arg2) is a generic function that divides **arg1** by **arg2** and returns the remainder. The argument types can be integer, real, or double precision. The result is the same data type as the arguments. If **arg2** is zero, the result is undefined.

Examples:

Function Call	Value Returned to *a*
a = MOD(30,13)	4
a = MOD(30.0, 13.0)	4.0
a = MOD(5, -3)	2
a = MOD(-5, 3)	-2
a = MOD(-5, -3)	-2

The specific function names are MOD for INTEGER*4 arguments, AMOD for real arguments, DMOD for double precision arguments, and HMOD for INTEGER*2 arguments.

MVBITS

MVBITS(arg1, arg2, arg3, arg4, arg5) is a procedure that moves **arg3** bits starting from position **arg2** of **arg1** to position **arg5** of **arg4**. The portion of **arg4** not affected by the movement of bits remains unchanged. All arguments are integer expressions, except **arg4**, which must be an integer variable or array element. Arguments **arg1** and **arg4** can be the same numeric storage unit. The value of **arg2 + arg3** cannot exceed the bit length of **arg1** and the value of **arg5 + arg3** cannot exceed the bit length of **arg4**.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0. Figure 5-1 shows how the MVBITS procedure works.

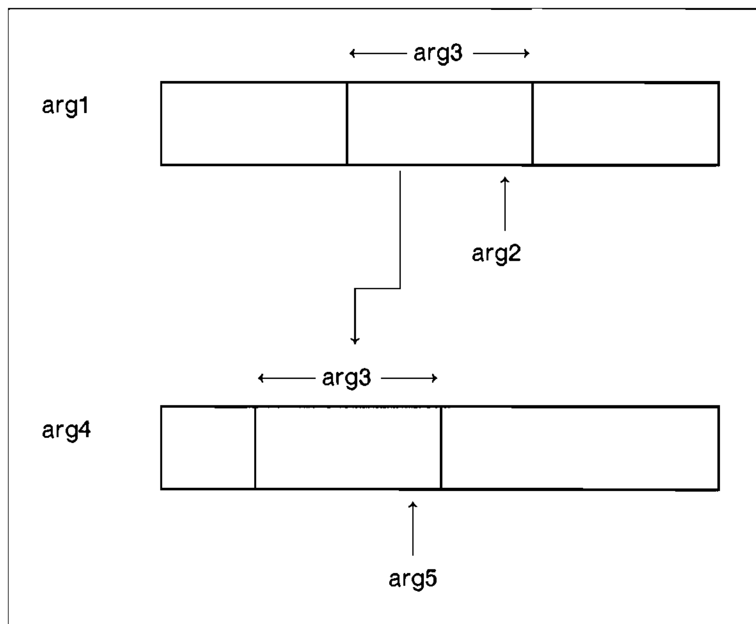


Figure 5-1. The MVBITS Procedure

The specific procedure names are MVBITS for INTEGER*4 arguments and HMVBITS for INTEGER*2 arguments.

Examples:

```
      PROGRAM move_bits

      INTEGER*4 int, int2

C   To clear the 12 low-order bits of int:
      CALL MVBITS(0, 0, 12, int, 0)

C   Read a character into int2 in R1 format:
      READ(5, '(R1)') int2

C   Copy character to the remaining bits of int2:
      DO j = 1, 3
         CALL MVBITS(int2, 0, 8, int2, j*8)
      END DO

      END
```

NINT

NINT(arg) is a generic function that returns the nearest integer. The argument can be real or double precision; the result is integer. If arg exceeds the largest integer allowed, the result will be undefined. If the argument is positive or zero, the result is equal to INT (arg + 0.5). If the argument is negative, the result is equal to INT (arg - 0.5).

Examples:

Function Call	Value Returned to *i*
i = NINT(123.456)	123
i = NINT(123.987)	124
i = NINT(123.5)	124
i = NINT(-123.456)	-123
i = NINT(-123.987)	-124

The specific function names are NINT for real arguments and IDNINT for double precision arguments.

NOT

NOT(arg) is a generic function that returns the bitwise complement of an integer argument. The result is integer.

Examples:

Function Call	Value Returned to *i*
i = NOT(1)	-2
i = NOT(0)	-1
i = NOT(5)	-6
i = NOT(-1)	0

The preceding examples show the use of twos complement arithmetic.

The specific function names are NOT for INTEGER*4 arguments and HNOT for INTEGER*2 arguments.

REAL

REAL(arg) is a generic function that converts an argument to a real number. The argument can be integer, real, double precision, or complex. If **arg** is real, the result is equal to **arg**. If **arg** is integer or double precision, the result is as much precision of the significant part of **arg** as a real item can contain. For a complex argument **(ar, ai)**, the result is **ar**.

Examples:

Function Call	Value Returned to *r*
r = REAL(5)	5.0
r = REAL(5.5)	5.5
r = REAL(5.55555D2)	555.555
r = REAL(var)	5.5, where var = (5.5, 5)

The specific function names are FLOAT for integer arguments and SNGL for double precision arguments.

SIGN

SIGN(arg1, arg2) is a generic function that transfers the sign from one numeric value to another. SIGN **(arg1, arg2)** returns the magnitude of **arg1** with the sign of **arg2**. The arguments can be integer, real, or double precision. The result is the same data type as the argument. The result is |**arg1**| if **arg2** is positive or zero, and -|**arg1**| if **arg2** is negative.

Examples:

Function Call	Value Returned to *a*
a = SIGN(45.84, -133.0)	-45.84
a = SIGN(45.84, 133.0)	45.84
a = SIGN(-45.84, -133.0)	-45.84

The specific function names are SIGN for real arguments, DSIGN for double precision arguments, ISIGN for INTEGER*4 arguments, and HSIGN for INTEGER*2 arguments.

SIN

SIN(arg) is a generic function that returns the sine of the argument. The argument is expressed in radians and is real, double precision, or complex. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = SIN(0.0)	0.0
a = SIN(1.5708)	1.0
a = SIN(0.0628)	0.06275873

The specific function names are SIN for real arguments, CSIN for COMPLEX*8 arguments, DSIN for double precision arguments, and ZSIN for COMPLEX*16 arguments.

SINH

SINH(arg) is a generic function that returns the hyperbolic sine of a real or double precision argument. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = SINH(0.0)	0.0
a = SINH(1.5708)	2.3013079

The specific function names are SINH for real arguments and DSINH for double precision arguments.

SQRT

`SQRT(arg)` is a generic function that returns the square root of a real, double precision or complex argument. The result is the same data type as the argument. The argument cannot be negative for real and double precision values.

Examples:

Function Call	Value Returned to *a*
`a = SQRT(9.0)`	3.0
`a = SQRT(49.0D0)`	7.0
`a = SQRT(var)`	(5.0, 0.0), where var = (25, 0.0)

The specific function names are SQRT for real arguments, CSQRT for COMPLEX*8 arguments, DSQRT for double precision arguments, and ZSQRT for COMPLEX*16 arguments.

TAN

`TAN(arg)` is a generic function that returns the tangent of the argument. The argument is expressed in radians and is real, double precision, or complex. The value of |**arg**| must be less than or equal to the maximum number allowed on your system and not close to $\pm(2n+1)*\pi/2$, where n is an integer. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
`a = TAN(3.0)`	-0.1425465
`a = TAN(1.0)`	1.5574077

The specific function names are TAN for real arguments, CTAN for COMPLEX*8 arguments, DTAN for double precision arguments, and ZTAN for COMPLEX*16 arguments.

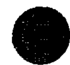
TANH

TANH(arg) is a generic function that returns the hyperbolic tangent of a real or double precision argument. The result is the same data type as the argument.

Examples:

Function Call	Value Returned to *a*
a = TANH(3.0)	0.9950548
a = TANH(1.0)	0.7615942

The specific function names are TANH for real arguments and DTANH for double precision arguments.



Chapter 6
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. You should 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 four areas:

- Compile time
- Run time
- Code space
- Data space

For system-dependent suggestions in this manual, refer to the appropriate appendix for your system.

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 the word size of your machine. If you are on a 32-bit machine, use the defaults of INTEGER*4 and LOGICAL*4. If you are on a 16-bit machine, change the defaults to INTEGER*2 and LOGICAL*2 with the $SHORT compiler option.

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:

Integer	(Fastest)
Real	
Double Precision	(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

Mixed-mode expressions should be avoided. 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 porting to different systems because the internal representation of y might vary between systems.

Use Statement Functions

Use statement functions instead of short function subroutines. This eliminates the overhead involved with loading parameters and avoids a procedure call for the subroutine 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 loop 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 3 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. 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.

Code Space Efficiency

The suggestions below help to decrease the amount of space needed to store your program.

Use Function Subroutines

Use function subroutines 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 subroutine 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
```

If you have a cross reference facility for your compiler, the facility can be used to locate the redundant and unused variables.

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.

Group Variables

When using COMMON and EQUIVALENCE statements, group variables of the same type. This prevents wasted space due to differences in data type alignment.

Use Short Integer and Logical Data

Use the $SHORT compiler option so integer and logical data use 16 bits instead of 32 bits. However, be aware that some constructs require four-byte types. Among these are all integer and logical I/O specifiers whose values are set by the I/O library (such as the IOSTAT specifier and most INQUIRE statement specifiers), and integer variables in ASSIGN statements. Also, if you are on a 32-bit machine, you might decrease performance efficiency by using 16-bit data.

Chapter 7
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.

See the system-specific appendix in this manual for additional portability topics.

In general, methods for program portability fall into these four categories:

- Restricting programs to features and statements that are a part of HP FORTRAN 77.
- 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.

Using these methods does not guarantee that your program will compile directly, 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 document. 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 FORTRAN 77 Reference Manual.

Do not include any feature that is not a part of the standard. When moving programs from a non-HP system or from FORTRAN 77 on 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 Military Standard or HP extensions with warnings. Thus, 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. See the system-dependent appendix in this manual for the compiler directives that help identify nonstandard features. See the system-dependent appendix of the *HP FORTRAN 77 Reference Manual* for more details on the ANSI directive.

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. Thus, by using the ANSI directive or a system-dependent directive, it is easy to identify where features are used 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 would 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 data 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 implicit 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 thus should not create any problems. Refer to the system-dependent appendix in the *HP FORTRAN 77 Reference Manual* for more information on the LONG and SHORT directives.

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. 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 integers; 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. Thus, 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 Military Standard 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. Using IMPLICIT NONE also flags typographical errors of variable names.

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 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, 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. Unfortunately, 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 your data in two ways:

First, if your program is small, has very few variables, and does not have common blocks, use assignment statements to improve documentation and readability.

Second, 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 Reference Manual* 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 before 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 parameters 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 system-specific appendix in this 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. All common blocks of the same name should have the same internal structure. Alignment problems are avoided by a minimal use of the EQUIVALENCE statement.

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 can also be easily changed. This section describes some guidelines of 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 one space per nesting level, and double spaces between logical sections 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. Thus, 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 1. In addition, an exclamation point (!) identifies a comment. 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; see the *HP FORTRAN 77 Reference Manual* 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 gives 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. Therefore, if you apply the guidelines given above, your programs will be more 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 thus 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.

USING FORTRAN 77 ON THE HP 3000

APPENDIX A

This appendix describes how to use FORTRAN 77 on the MPE operating system. Included is information on using files, passing parameters, accessing data, debugging, writing efficient programs, programming for portability, and interfacing with other languages, system intrinsics, and the HP 3000 subsystems.

NOTE

In this appendix, the term "FORTRAN 77" implies "HP FORTRAN 77".

FORTRAN 77 FILE OPERATIONS ON THE HP 3000

With FORTRAN 77, the OPEN statement gives you more control over file connection and file characteristics than under older versions of FORTRAN. The OPEN statement is translated by the run-time library to call the MPE FOPEN intrinsic. The options specified in the OPEN statement are implemented by specifying the corresponding options in the FOPEN intrinsic. The options in the OPEN statement do not override the characteristics of an existing file. A :FILE command pertaining to the file takes precedence over the FOPEN arguments requested. Any FOPEN options not specified by FORTRAN 77 are supplied with the file system defaults. This section explains the characteristics requested by the OPEN statement processor, which are reflected primarily in new files created by FORTRAN 77 programs.

The options specified in the OPEN statement determine the values of the FOPTIONS and AOPTIONS arguments to the FOPEN intrinsic called by the OPEN statement processor. The default FILESIZE (number of records) and NUMEXTENTS (number of extents allowed) are 4096 and 32, respectively. Both values are four times the MPE default values to eliminate the need for :FILE equations when a file of more than 1023 records is needed. No more disc space is allocated for small files than if the MPE defaults were used.

Predefined Units and Files

The predefined I/O units for FORTRAN 77 are units 5 and 6. These units are opened by the I/O library before the first I/O statement is executed. Unit 5 is opened as formal file designator FTN05, which defaults to $STDINX. ($STDIN is not the default to avoid logging off if an input line of :EOF is entered, and to allow 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 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 files.

For example, using the equation

```
:FILE FTN06 = OUTFILE
```

with the FORTRAN 77 statement

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

causes the output to be written to file OUTFILE instead of to the terminal.

FTN05

Unit 5 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, which 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. These opens fail if exclusive access was requested.

FTN06

Unit 6 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, which 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.

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 not overridden by the characteristics of the corresponding device or file.

All files connected by the OPEN statement processor besides FTN05 and FTN06 are opened by FOPEN with AOPTIONS of 4, requesting read/write access. This is done because FORTRAN 77 does not have syntax to imply that a file being opened is used only for reading or writing. Therefore, the system must be prepared for any arbitrary mix of READ and WRITE statements. 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. These requests report a FILE SYSTEM ERROR.

STATUS = 'NEW'

When the OPEN statement specifies STATUS='NEW', the FORTRAN standard requires the implementation to ensure that the referenced file does not exist. FORTRAN 77 does this by first attempting to open the file as an MPE OLD or TEMP file. If the file exists, the file is closed and an error reported. Normally, the first FOPEN fails because no file exists and the FOPEN is tried again with FOPTIONS requesting an MPE 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', an FOPEN specifies that only the MPE OLD or TEMP file domains are searched. The job temporary (TEMP) 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 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 attempted again with NEW status.

FORM = 'UNFORMATTED' and FORM = 'FORMATTED'

The FORM option specifies the types of transfers that can be performed on the file. FORM='UNFORMATTED' implies that only unformatted (binary) transfers are done, so an MPE BINARY file is requested in the FOPEN. For 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 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.

ACCESS = 'SEQUENTIAL'

Sequential READ and WRITE statements can have I/O lists of varying lengths; in fact, the ANSI standard does not specify an upper limit to the length of a sequential record. The only way to efficiently implement varying lengths is to use MPE variable record length files as the default sequential file type (FOPTIONS bit 8 cleared to zero, bit 9 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 default of the configured physical record size of the device is to be used (256 bytes on disc files). Also, on variable record length files created by the OPEN statement, the MPE 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 does this for you, this is clearly is 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 8 and 9 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 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 is used to request 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.

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 status is not specified in the CLOSE statement, named files default to KEEP (kept as an MPE permanent file) and scratch files default to DELETE. If you attempt to save a scratch file, an error results.

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 job TEMP 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 TEMP domain. If the second FCLOSE fails, an error results.

Terminating the Program

All files remaining open when your program terminates are closed either by the FORTRAN 77 library or by MPE. 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; these errors are detected by the FORTRAN 77 library, the HP 3000 compiler library, or by MPE.

Errors found by the FORTRAN 77 library (primarily I/O errors) that are not trapped by the ERR or IOSTAT specifiers cause an error message to be printed. If no errors occur, files are closed in a normal termination, as described above.

Errors detected by the compiler library (primarily math function errors) or by MPE cause error messages to be printed. However, any opened files are closed by MPE, not FORTRAN 77. When closed by MPE, an FCLOSE with MPE 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 causes files newly created to be deleted.

Carriage Control Files

The preconnected unit 6 (FTN06) is opened with carriage control (CCTL). When initially opening empty files (EOF points to 0), the CCTL bit and the device type are checked. The appropriate FWRITE is then performed to set the file for prespacing.

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.

Disc Files

If a carriage control disc 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. FORTRAN/3000 programs that previously did this explicitly with a WRITE statement still work as designed. For example, the statement

```
WRITE(12,'("A")')
```

is valid.

: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)
```

cause the output to be written to disc 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 using 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 1. However, the value 1 wastes tape because the inter-record 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
```

cause 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 still supported by FORTRAN 77. All the options of the FORTRAN/3000 version are supported on FORTRAN 77, including those operations commonly used for magnetic tape handling.

Using the FSET Procedure

The FSET procedure changes the MPE/3000 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)
```

Parameters

`unit` is a positive single word integer or an integer variable (INTEGER*2) that specifies the file table entry for which the change is to be made.

`newfile` is a positive single word integer or an integer variable (INTEGER*2) that specifies the new MPE file number that is to be assigned to the `unit` specified above.

`oldfile` is a single word variable (INTEGER*2) to which the procedure returns the old value of the file number that was assigned to the `unit` specified above.

NOTE

All arguments to FSET are INTEGER*2 for backwards compatability to FORTRAN/3000.

The following program shows how to use the FSET procedure to assign a FORTRAN logical unit number.

```
 0     1.000    $WARNINGS OFF
 0     2.000    $SHORT
 1     3.000          PROGRAM fset_example
 1     4.000    C
 1     5.000    C
 1     6.000    C  FOPEN, FSET, AND FCLOSE EXAMPLE
 1     7.000    C
 2     8.000          IMPLICIT NONE
 3     9.000          INTEGER filenumber,oldnum !$SHORT implies INTEGER*2
 4    10.000          SYSTEM INTRINSIC FOPEN,FCLOSE
 5    11.000          CHARACTER buffer*72,filename*16
 6    12.000          PARAMETER (filename = 'MAILLIST ')
 6    13.000
 7    14.000          filenumber = FOPEN(filename,1B,105B)
 8    15.000          IF (CCODE())30,10,30
 8    16.000
 8    17.000    C  Call FSET to assign the FORTRAN unit number 5 to
 8    18.000    C  "filenumber"
 8    19.000
 9    20.000     10   CALL FSET(5,filenumber,oldnum)
10    21.000          PRINT *,'Old file number = ',oldnum
11    22.000          PRINT *,'FOPEN number = ',filenumber
12    23.000     20   READ(5, '(A72)' ,END=40)buffer       ! Read to EOF
13    24.000          WRITE(6,100)buffer(1:19)
14    25.000     100  FORMAT(T2,A20)
15    26.000          GO TO 20
16    27.000     30   PRINT *,'COULD NOT OPEN FILE'
17    28.000          STOP
17    29.000
17    30.000    C  Close file
17    31.000
18    32.100     40   CALL FCLOSE(filenumber,1B,0B)
19    33.000          IF (CCODE())50,60,50
20    34.000     50   PRINT *,'Could not close file'
21    35.000          STOP
22    36.000     60   PRINT *,'File closed successfully'
23    37.000          STOP
24    38.000          END
```

The output of the program looks like this:

```
 Old file number =  4
 FOPEN number =  3
  SMILEY     FACE
  MICKEY     MOUSE
  SLIM       JIM
  CHARITY    BELL
  DONALD     DUCK
  JOE        SMOE
  CLAIRE     PLIMSOL
  INDIANA    JONES
  JAKE       FAKE
 File closed successfully
```

Using the FNUM Procedure

The FNUM procedure can be called from a FORTRAN 77 program to extract the MPE/3000 system file number assigned to a given FORTRAN 77 logical unit number from the file table; the procedure returns an INTEGER*2 value.

The FNUM procedure can be called from a FORTRAN 77 program as an external function as follows:

```
I = FNUM(unit)
```

Parameter

`unit` is a positive single word integer (INTEGER*2) that specifies the file table entry for which the MPE/3000 system file is to be extracted.

See the UNITCONTROL procedure description below for an example of using the FNUM procedure.

Using the UNITCONTROL Procedure

The UNITCONTROL procedure allows an HP FORTRAN 77 program to request several actions (see below) for any FORTRAN 77 logical unit.

The UNITCONTROL procedure is called as follows:

```
CALL UNITCONTROL(unit,opt)
```

Parameters

`unit` is a positive single word integer (INTEGER*2) that specifies the unit number of the file to be used. (INTEGER*2 is limited to a maximum of 32767.)

opt is a single word integer (INTEGER*2) that specifies one of the following options:

```
-1:  REWIND (but don't close file)
 0:  BACKSPACE
 1:  ENDFILE (write an EOF mark)
 2:  SKIP BACKWARD TO A TAPE MARK
 3:  SKIP FORWARD TO A TAPE MARK
 4:  UNLOAD TAPE AND CLOSE THE FILE
 5:  LEAVE TAPE AND CLOSE THE FILE
 6:  CONVERT FILE TO PRESPACING
 7:  CONVERT FILE TO POSTSPACING
 8:  CLOSE FILE
```

NOTE

Instead of using the options -1, 0, 1, and 8, the statements REWIND, BACKSPACE, ENDFILE, and CLOSE should be used. These statements are part of the FORTRAN 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.

```
0     1.000   $WARNINGS OFF
0     2.000   $SHORT
1     3.000         PROGRAM unit_fnum_ex
1     4.000   C
1     5.000   C  Example program using FNUM and UNITCONTROL
1     5.100   C
2     6.000         IMPLICIT NONE
3     7.000         SYSTEM INTRINSIC MERGE
4     8.000         CHARACTER buffer*72
5     9.000         INTEGER keys(6),infiles(2)
6    10.000         LOGICAL failure
6    11.000
6    12.000   C  Merge two sorted files (MAIL1 (unit 20) and MAIL2
6    13.000   C  (unit 21) into a third file (MAIL3 unit 22))
6    14.000
6    15.000   C  Open all files
6    16.000
7    17.000          OPEN(20,FILE='mail1',STATUS='OLD',ERR=200)
8    18.000          OPEN(21,FILE='mail2',STATUS='OLD',ERR=300)
9    19.000          OPEN(22,FILE='mail3',STATUS='NEW',ERR=400)
9    20.000
```

```
    9     21.000      C  Establish keys for SORT - major at 11 for 9 bytes
    9     22.000      C  (LAST NAME) and minor at 1 for 10 bytes (FIRST NAME)
    9     23.000
   10     24.000            keys(1) = 11
   11     25.000            keys(2) = 9
   12     26.000            keys(3) = 0
   13     27.000            keys(4) = 1
   14     28.000            keys(5) = 10
   15     29.000            keys(6) = 0
   15     30.000
   15     31.000      C  Establish MPE/3000 filenumbers for input files (MAIL1
   15     32.000      C  and MAIL2) by referencing FNUM procedure
   15     33.000
   16     34.000            infiles(1) = FNUM(20)
   17     35.000            infiles(2) = FNUM(21)
   17     36.000
   17     37.000      C  Merge the files.  Note: Be careful when using FORTRAN
   17     38.000      C  logicals with SPL logical
   18     39.000            CALL MERGE(2,infiles,FNUM(22),0,2,keys,,,,,,failure)
   19     40.000            IF (failure) STOP 'Merge failed'
   19     41.000
   19     42.000      C  Display the new merged file
   19     43.000
   20     44.000            REWIND 22
   21     45.000       20   READ(22,'(A72)',END=30) buffer
   22     46.000            WRITE(6,100)buffer
   23     47.000       100  FORMAT(T2,A)
   24     48.000            GO TO 20
   24     49.000
   24     50.000      C  Call UNITCONTROL to close files MAIL1, MAIL2, and MAIL3.
   24     51.000      C  Same as using the CLOSE statement.
   24     52.000
 PAGE     2


   25     53.000       30   CALL UNITCONTROL(20,8)
   26     54.000            CALL UNITCONTROL(21,8)
   27     55.000            CALL UNITCONTROL(22,8)
   28     56.000            STOP
   28     57.000
   29     58.000       200  PRINT *,'Could not open file - MAIL1'
   30     59.000            STOP
   31     60.000       300  PRINT *,'Could not open file - MAIL2'
   32     61.000            STOP
   33     62.000       400  PRINT *,'Could not open output file - MAIL3'
   34     63.000            END
```

The output of the program looks like this:

```
PLAINS    ANTELOPE 201 OPENSPACE 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  OPENSPACE   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
```

PASSING RUN COMMAND PARAMETERS

A maximum of two input parameters from a program's RUN command can be passed to a program. One parameter must be a CHARACTER*(*) data type and the other an INTEGER*2 type. For example, if you want to pass two parameters to the program `test`, where one parameter is a character string and the other is an integer, you need these statements:

```
PROGRAM test(p1,p2)
CHARACTER*(*) p1
INTEGER*2 p2
```

In the program's RUN command, the character parameter is identified with the INFO string and the INTEGER parameter is identified with the PARM word. Data is passed to the program `test` with the following RUN command:

```
:RUN test;INFO="infile";PARM=3
```

The program would then have p1 and p2 initialized as if they appeared in these assignment statements:

```
p1='infile'
p2=3
```

The order of the parameters does not matter in either the program statement or in the program's RUN command.

ACCESSING DATA

The three modes of accessing data on HP 3000 systems are:

- Direct
- Indirect
- By Descriptor

These modes are summarized in Table A-1.

Table A-1. Summary of Accessing Modes

MODE	VARIABLE
Direct	Simple variables less than or equal to eight bytes
Indirect	Variables in COMMON Variables in a SAVE, DATA, or EQUIVALENCE statement All other variables greater than eight bytes Parameters not of type CHARACTER
Descriptors	Character string parameters

The descriptor is composed of two words: one word contains the address of the data and the other contains the length.

DEBUGGING FORTRAN 77 PROGRAMS

This section shows two ways to debug a FORTRAN 77 program. The first method uses MPE debug, and the second uses the $RANGE directive.

MPE Debug

To debug a FORTRAN 77 program on the HP 3000, follow these steps:

1. Compile the program into a USL file with the $TABLES and $CODE_OFFSETS compiler directives ON and direct the listing to the line printer. The $TABLES directive lists each identifier and its stack location; the $CODE_OFFSETS directive shows the P register offset for each statement in a program unit.

2. Prepare the USL file with the PMAP option to a program file and direct the map to the line printer. The PMAP option indicates the program unit location within a code segment and is available through the :PREP command.

3. Run the program using the DEBUG facility. Set appropriate breakpoints by using the segment number and the code location from the PMAP combined with the statement offset from the CODE_OFFSET listing.

4. Resume program execution and when the breakpoint occurs, use the variable locations on the TABLES listing to display or manipulate the current variable values.

The example below shows a session that debugs a program using the steps listed above. The program with the error looks like this:

```
$TABLES ON,CODE_OFFSETS ON
      PROGRAM show_debug
      INTEGER*2 intarray(20),j
      PRINT *,'Begin program'
      DO 2 j=1,25
      intarray(j)=j
 2    ENDDO
      CALL printarray(intarray)
      STOP
      END


      SUBROUTINE printarray(integerarray)
      INTEGER*2 integerarray(20)
      WRITE(6,1)(integerarray(j),j=1,20)
 1    FORMAT(1X,I2)
      RETURN
      END
```

When the program above is run, a BOUNDS VIOLATION occurs, as shown below:

```
:run prgp


Begin program

ABORT :PRGP.FORTRAN.EX.%0.%160
PROGRAM ERROR #24 :BOUNDS VIOLATION

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
```

To use MPE debug, compile the program with the $TABLES and $CODE_OFFSETS compiler directives on. Direct the listing to the line printer to get a printout, as shown below. The highlighted lines in the output below are explained after the debug session.

```
:FILE LP;DEV=LP
:FTN prg,prgusl,*lp


 PAGE     1  HEWLETT-PACKARD   HP32116X.00.02
   HP FORTRAN 77  (C) HEWLETT-PACKARD CO. 1984   WED, DEC 19, 1984,  5:24 PM


     0     1.000      $TABLES ON,CODE_OFFSETS ON
     1     2.000            PROGRAM show_debug
     2     3.000            INTEGER*2 intarray(20),j
     3     4.000            PRINT *,'Begin program'
     4     5.000            DO 2 j=1,25
     5     6.000  1         intarray(j)=j
     6     7.000  1    2    ENDDO
     7     8.000            CALL printarray(intarray)
     8     9.000            STOP
     9    10.000            END
     0    11.000
     0    12.000

Name             Class            Type            Offset     Location
----             -----            ----            ------     --------

INTARRAY         Array (1 Dim)    Integer*2       Q+1        Local
J                Variable         Integer*2       Q+2        Local
PRINTARRAY       Subroutine
SHOW_DEBUG       Program
2                Stmt Label       Executable



                         C O D E   O F F S E T S

STMT   P-LOC     STMT   P-LOC     STMT   P-LOC     STMT   P-LOC     STMT   P-LOC
   3 000005         4 000026         5 000050         6 000061         7 000072
   8 000074         9 000077

     1    13                SUBROUTINE printarray(integerarray)
     2    14.000            INTEGER*2 integerarray(20)
     3    15.000            WRITE(6,1)(integerarray(j),j=1,20)
     4    16.000       1    FORMAT(1X,I2)
     5    17.000            RETURN
     6    18.000            END

Name             Class            Type            Offset     Location
----             -----            ----            ------     --------

INTEGERARRAY     Array (1 Dim)    Integer*2       Q-4        Argument
J                Variable         Integer*4       Q+1        Local
PRINTARRAY       Subroutine
1                Stmt Label       Format
```

```
                         C O D E   O F F S E T S

STMT  P-LOC    STMT  P-LOC    STMT  P-LOC    STMT  P-LOC    STMT  P-LOC
   3 000015       5 000076       6 000077


   NUMBER OF ERRORS =      0   NUMBER OF WARNINGS =    0
   PROCESSOR TIME       3173   ELAPSED TIME        13321
   NUMBER OF LINES =      18   LINES/MINUTE =        340
```

The USL file should now be prepared (prepped) with the PMAP option to get the entry of each program unit.

```
:FILE seglist=*lp
:PREP prgusl,prgp;pmap

PROGRAM FILE PRGUSL.FORTRAN.EX

SEG'                 0
   NAME             STT  CODE ENTRY SEG
   PRINTARRAY         1     0    13
   FTN_DO_I2IO        3               ?
   FTN_E_WSFE         4               ?
   FTN_S_WSFE         5               ?
   SHOW_DEBUG         2   100   101
   FTN_S_STOP         6               ?
   FTN_E_WSLE         7               ?
   FTN_S_WSLE        10               ?
   FTN_DO_CHIO       11               ?
   FTN_F_EXIT        12               ?
   TERMINATE'        13               ?
   SEGMENT LENGTH         230

PRIMARY DB          0    INITIAL STACK    10240    CAPABILITY        600
SECONDARY DB        0    INITIAL DL           0    TOTAL CODE        230
TOTAL DB            0    MAXIMUM DATA         ?    TOTAL RECORDS       6
ELAPSED TIME    00:00:03.443                  PROCESSOR TIME    00:00.635

END OF PREPARE
```

With the information obtained so far, you can find the line in the source code causing the error. Again, the error message that occurred when you ran the program looked like this:

```
ABORT :PRGP.FORTRAN.EX.%0.%160
PROGRAM ERROR #24 :BOUNDS VIOLATION
```

The message states that the program "PRGP.FORTRAN.EX" aborted at segment %0 and offset at %160 (these numbers are always in octal). Looking at the PMAP listing, you can see that there is only one

segment, SEG', starting at segment number %0. The PMAP tells you that the code of the procedure PRINTARRAY goes from %0 to %77 and the code for the main program SHOW_DEBUG goes from %100 to %230. Therefore, %160 is in the main program. The code offsets listing generated by the $CODE_OFFSETS compiler directive starts at the beginning (%0) for each program unit. So, if you take the location where the program aborts (%160) and subtract the location where the main program begins (%100), you get the abort location from the beginning (%0) of the main program. The result of %160 minus %100 is %60. If you look at the code offsets listing for the main program, you can see that %60 falls between P_LOCATION 50 and 61, which means that the program aborts at statement 5 in the source listing. Statement 5 assigns an integer into the array intarray. Notice that the program attempts to index the array from 1 to 25, even though the array is only 20 elements, thus causing a BOUNDS VIOLATION.

MPE debug can also check the value of variables in a program. For example, in the program above, you can check the value of the variable j at the end of the DO loop at each iteration. To do this, run the program with debug, set a breakpoint at source statement 6, and look at the location of j on the stack. The statements in the code offsets listing shows that statement 6 is at P location %61 and, because the main program begins at location %100 (refer to PMAP), the breakpoint will be set at %161. Referring to the tables listing, you can see that the variable j is stored at location Q+2. The following debug session looks at the value of j.

```
:run prgp;debug

*DEBUG* PRIV.0.101
?=100+61
 =161
?b 0.161:@
?r


Begin program
*BREAK* PRIV.0.161
?d q+2,1
Q+2       000002
?r


*BREAK* PRIV.0.161
?d q+2,1
Q+2       000003
?r


*BREAK* PRIV.0.161
?d q+2,1
Q+2       000004
?c@
?r


ABORT :PRGP.FORTRAN.EX.%0.%160
PROGRAM ERROR #24 :BOUNDS VIOLATION

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
```

The debug session above first sets a permanent breakpoint at %161. Then, the session resumes execution to reach the breakpoint, displaying location Q+2 (variable j). The program continues and breaks each time the breakpoint is reached, displaying Q+2 (this is done three times). Then, all breakpoints are cleared and the program resumes.

Breakpoints can be set at other locations and other variables can be displayed. Refer to the *MPE Debug/Stack Dump Reference Manual* for information on the debug commands.

$RANGE Directive

The example above can be debugged by using the $RANGE directive, as shown below:

```
:ftn fdebug

 PAGE     1  HEWLETT-PACKARD   HP32116A.00.00
   HP FORTRAN 77  (C) HEWLETT-PACKARD CO. 1984   THU, MAR 28, 1985, 10:57 AM


    0      1.000     $RANGE ON
    1      2.000           PROGRAM show_debug
    2      3.000           INTEGER*2 intarray(20),j
    3      4.000           PRINT *, 'Begin program'
    4      5.000           DO 2 j=1,25
    5      6.000  1        intarray(j)=j
    6      7.000  1    2   ENDDO
    7      8.000           CALL printarray(intarray)
    8      9.000           STOP
    9     10.000           END
    0     11.000
    1     12.000           SUBROUTINE printarray(integerarray)
    2     13.000           INTEGER*2 integerarray(20)
    3     14.000           WRITE(6,1)(integerarray(j),j=1,20)
    4     15.000       1   FORMAT(1X,I2)
    5     16.000           RETURN
    6     17.000           END

   NUMBER OF ERRORS =      0   NUMBER OF WARNINGS =    0
   PROCESSOR TIME    0: 0: 2   ELAPSED TIME       0: 0:17
   NUMBER OF LINES =      17   LINES/MINUTE =        465


END OF PROGRAM
:prep $oldpass,prg

END OF PREPARE
```

```
:run prg


Begin program
*** FORTRAN RANGE ERROR 950: SUBSCRIPT OR SUBSTRING OUT OF BOUNDS AT STATEMENT N
UMBER 5
      ***      STACK  DISPLAY      ***


                        S=000211    DL=177044    Z=010240
          Q=000215 P=012057  LCST= G003  STAT=U,1,1,L,0,0,CCL   X=000016

          Q=000077 P=012222  LCST= G003  STAT=U,1,1,L,0,0,CCE   X=000023
          Q=000071 P=000210  LCST=  000  STAT=U,1,1,L,0,0,CCG   X=000023

ABORT :PRG.A0000.F773000.%0.%210:GRSL.%3.%12067
PROGRAM ERROR #19 :PROGRAM QUIT

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
```

WRITING EFFICIENT PROGRAMS

This section lists methods for improving program efficiency on the HP 3000; additional methods are discussed in Chapter 6.

COMPILE TIME EFFICIENCY

- Allocate FTN.PUB.SYS and FTN2.PUB.SYS to make the compiler load faster.
- Break up large program units into smaller subprograms. Smaller program units are more efficient because the compiler does not have to use its less efficient secondary storage area.

RUN-TIME EFFICIENCY

- Use MPE intrinsic I/O instead of FORTRAN READ and WRITE statements. The FORTRAN statements generate several procedure calls for each statement. However, using MPE intrinsics makes your program system-dependent and difficult to port.
- When possible, use DO loops instead of DO WHILE loops. The code generated to evaluate the loop counter is more efficient for DO loops if the loop counter is INTEGER*2 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 address must be found first, and then the element at that address found.
- Use $MORECOM only in programs in which you cannot avoid a large number of COMMON variables. Using $MORECOM introduces an additional level of indirection when addressing common variables.

DATA SPACE EFFICIENCY

- By changing the order of the variables, you might be able to increase the number of variables allowed in a COMMON block without using the less efficient MORECOM option. The MPE Segmenter sets up a table that contains addresses of all the variables in the common block. The maximum number of addresses allowed is 254 when MORECOM is not used. Normally there is one address for each common block variable.

  For example, there are three addresses for a common block containing the variables c(3), a, and b. However, if the order of the variables is changed to a, b, c(3) (the array follows the simple variables), only two addresses are required because array addresses are set to point to the *zero*th element of the array. But, because FORTRAN arrays start with the first element, that would be the same location as b when the variables are listed a, b, c(3). Therefore, a single pointer is shared for variable b and array c.

CODE SPACE EFFICIENCY

- If a long character constant is used more than once, assign the constant to a variable and reference the variable instead of the constant. Otherwise, the constant is placed into your object file each time the constant is used. Of course, using the variable expands your data space, so this should not be done if you are low on data space.

PROGRAMMING FOR PORTABILITY

This section describes methods to help make your programs portable.

Identify 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 ANSI ON mentioned in Chapter 7 and flags all non-ANSI features with warning messages.

The STANDARD_LEVEL HP option issues warning messages only for features that are not a part of the FORTRAN 77 standard, and gives a quick assessment of the non-HP FORTRAN 77 features in your programs.

The default STANDARD_LEVEL SYSTEM option issues no special warning messages for nonstandard features or messages for system-specific features.

For more details on the syntax and semantics of the ANSI and STANDARD_LEVEL compiler directives, refer to the appendix in the *HP FORTRAN 77 Reference Manual* for your system.

Avoid Inconsistencies

On the HP 3000, 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 linker, allowing the loader or linker to

check for inconsistencies in type, length, and number of parameters. See the appropriate appendix in the *HP FORTRAN 77 Reference Manual* for a complete description.

Use Comments

When you have sections of code that are specific to the HP 3000, you can identify the lines as comment lines and make appropriate source changes. Place a C or * in the first column of the system-specific code to indicate that the statements are to be treated as comment lines. This keeps the code in the program, but the statements are not executed. When the program is run on the HP 3000 again, remove the C or *.

Use Conditional Compilation Directives

Use the conditional compilation directives, as summarized in Table A-2.

Table A-2. Summary of Conditional Compilation Directives

DIRECTIVE	DESCRIPTION
$ELSE	Used with the IF directive; marks the beginning of the ELSE block of code.
$ENDIF	Ends the $IF directive.
$IF	Conditionally compiles blocks of code.
$SET	Assigns values to identifiers used in IF directives.

For example, the partial program below uses the HP 3000 system intrinsics if the program is run on the HP 3000, and uses FORTRAN I/O for portable code.

```
      PROGRAM sys3000

$SET (OS_MPE = .TRUE.)

      CHARACTER buffer(80)

$IF 'OS_MPE'
      SYSTEM INTRINSIC FREAD, FWRITE
$ENDIF
      .
      .
      .
$IF 'OS_MPE'
C  MPE 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
```

INTERFACING WITH OTHER LANGUAGES

This section describes how to call routines coded in SPL (Systems Programming Language), FORTRAN/3000, PASCAL, or COBOL from a FORTRAN 77 program, and how to call a FORTRAN 77 subroutine or function from another language.

If you call other languages from FORTRAN 77, the actual parameters of the procedure or function must match the formal parameters of the external procedure or function. The CHECK_ACTUAL_PARM compiler option lets you determine the level of the checking information placed in the USL for the segmenter when performing a PREP or ADDSL. See the *HP FORTRAN 77 Reference Manual* for more information on the CHECK_ACTUAL_PARM compiler option.

When calling FORTRAN 77 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 lets you determine the checking information placed in the USL file for the segmenter when performing a PREP or ADDSL. See the *HP FORTRAN 77 Reference Manual* for more information on the CHECK_FORMAL_PARM compiler option.

FORTRAN 77 uses the DL-DB area of the stack to store its file table. If the non-FORTRAN 77 program uses subsystems, such as VPLUS or DSG, the program must use the language code 5. Using the language code 5 permits the subsystem, which also uses the DL-DB area, to call the procedures GETHEAP and RTNHEAP.

All parameters passed to a FORTRAN 77 subprogram must be passed by reference. By default, FORTRAN 77 passes parameters by reference. FORTRAN 77 allows parameters to be passed by value to invoke non-FORTRAN 77 program units that allow passing arguments by value. To pass parameters by value, the ALIAS compiler directive must be used to indicate how each of the parameters is passed. You must use the language code option of the ALIAS command when passing character strings to Pascal, FORTRAN, and SPL, and when passing logical variables to Pascal. For programming examples, see the Pascal section in this appendix.

FORTRAN 77 does not allow arrays to be passed by value. Therefore, you cannot interface a FORTRAN 77 program to 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, an empty parameter list, ( ), must be given.

SPL/3000

SPL (Systems Programming Language) for the HP 3000 is a high-level, machine dependent programming language used for developing compilers, operating systems, subsystems, monitors, and supervisors.

Calling SPL/3000 from FORTRAN 77

You must use the language code option of the ALIAS compiler directive if you call an external SPL routine from a FORTRAN 77 program. The ALIAS directive defines the procedure or function and matches the FORTRAN 77 types of the actual parameters or function type with the SPL types of the external parameters or function result type. Table A-3 lists the corresponding FORTRAN 77 and SPL types.

Table A-3. FORTRAN 77 and SPL/3000 Types

FORTRAN 77 Type	SPL/3000 Type
INTEGER, INTEGER*4	DOUBLE INTEGER
INTEGER*2	INTEGER
REAL	REAL
REAL*8, DOUBLE PRECISION	LONG
CHARACTER*n	BYTE ARRAY arrayname(0:n-1)
LOGICAL, LOGICAL*4	LOGICAL ARRAY arrayname(0:1)
LOGICAL*2	LOGICAL
COMPLEX, COMPLEX*8	REAL ARRAY arrayname(0:1)
COMPLEX*16, DOUBLE COMPLEX	LONG ARRAY arrayname(0:1)

SPL cannot accept value parameters of any array type. SPL supports only one-dimensional arrays, so multi-dimensional arrays passed by FORTRAN 77 are linearized in column-major order. Arrays are passed between SPL and FORTRAN 77 programs by supplying an array element as the actual parameter. The address of the first element of an SPL array points to the *zero*th value, while the address of the first element of a FORTRAN 77 array points to the first value.

SPL procedures can be declared as OPTION VARIABLE, which indicates that some parameters do not have to be supplied. Because this feature does not exist in FORTRAN 77, you must supply an extra logical value argument that is appended to the complete argument list. This value consists of one or more 16-bit parameter masks. This extra parameter serves as a "bit map", with each bit representing a parameter in the parameter list. The rightmost bit represents the last parameter in the list, the second rightmost bit the next-to-last parameter, and so on. Each "on" bit (bit=1) indicates that the corresponding parameter is actually being passed. Each "off" bit (bit=0) indicates a parameter is being supplied a dummy value. This is shown in the following examples.

When passing the FORTRAN 77 types LOGICAL, LOGICAL*4, CHARACTER, COMPLEX, and DOUBLE COMPLEX to SPL, set the CHECK__ACTUAL__PARM compiler option to 2 or less in the calling FORTRAN 77 program unit, or use OPTION CHECK 2 or less in the called SPL procedure.

Example

FORTRAN 77 program:

```
$SHORT
      PROGRAM fortprog
      IMPLICIT NONE
      CHARACTER str*20,longmsg*50
      INTEGER*2 length
$CHECK_ACTUAL_PARM 2
$ALIAS splproc SPL(%val,%ref,%ref,%val)
      length=LEN(str)
      str='Pass this to SPL'
      longmsg=' '
      PRINT *,str
      CALL splproc(length,str,0,6B)
      PRINT *,str
      CALL splproc(length,str,longmsg,7B)
      PRINT *,longmsg
      END
```

External SPL/3000 procedure:

```
$CONTROL SUBPROGRAM
BEGIN
PROCEDURE splproc(length,str,longmsg);
VALUE length;
INTEGER length;
BYTE ARRAY str,longmsg;
OPTION VARIABLE;
BEGIN
logical pmask=q-4;                  ! option variable bit mask
define longmsg'there=pmask#;
IF longmsg'there THEN               ! Longmsg exist
   BEGIN
   MOVE longmsg := str,(length);
   MOVE longmsg(length) := " and back";
   END
ELSE
   move str := "back";
END;
END.
```

Calling FORTRAN 77 from SPL/3000

You must use an SPL EXTERNAL procedure declaration if you call a FORTRAN 77 procedure or function from an SPL program. The procedure declaration provides parameter declarations compatible with the FORTRAN 77 types of the external parameters. See the section "Calling FORTRAN 77 from Pascal" in this appendix for more details. Table A-3 shows the FORTRAN 77 and SPL type correspondences.

An SPL program cannot pass any parameters by value to a FORTRAN 77 subroutine or function. FORTRAN 77 generates FORTRAN 77 type checking information in the USL file for FORTRAN 77 subroutines or procedures.

Example

SPL/3000 program:

```
BEGIN
   LOGICAL ARRAY chr(0:9) := "Add these 2 numbers:";
   BYTE ARRAY bchr(*) = chr;
   INTEGER sint:=15, sint2:=25, len, sum;

   INTRINSIC PRINT, ASCII;
   INTEGER PROCEDURE fort(sint,sint2);
      INTEGER sint,sint2;
      OPTION EXTERNAL;

   PRINT(chr,-20,0);
   PRINT(chr,-ASCII(sint,10,bchr),0);
   PRINT(chr,-ASCII(sint2,10,bchr),0);

   sum:=fort(sint,sint2);
   MOVE chr:="Sum of two numbers: ";
   PRINT(chr,-20,0);
   PRINT(chr,-ASCII(sum,10,bchr),0);
END.
```

External FORTRAN 77 function:

```
      INTEGER*2 FUNCTION fort(sint,sint2)
      IMPLICIT NONE
      INTEGER*2 sint,sint2

      fort = sint + sint2
      END
```

FORTRAN/3000

FORTRAN/3000 is the ANSI (X3.9-1966) version of FORTRAN for the HP 3000 computer.

Calling FORTRAN/3000 from FORTRAN 77

The types of the FORTRAN 77 actual parameters must match the FORTRAN/3000 formal parameters if you call a FORTRAN/3000 routine from a FORTRAN 77 program. Table A-4 lists FORTRAN 77 types and the corresponding FORTRAN/3000 types.

Table A-4. FORTRAN 77 and FORTRAN/3000 Types

FORTRAN 77 Type	FORTRAN/3000 Type
INTEGER	INTEGER*4
INTEGER*2	INTEGER
REAL	REAL
REAL*8, DOUBLE PRECISION	DOUBLE PRECISION
CHARACTER*n	CHARACTER*n
LOGICAL, LOGICAL*4	Does Not Exist
LOGICAL*2	LOGICAL
COMPLEX, COMPLEX*8	COMPLEX
COMPLEX*16, DOUBLE COMPLEX	Does Not Exist

FORTRAN 3000 can access a FORTRAN 77 COMMON area only if compiled into the same USL file. It is also possible to intermix I/O between FORTRAN 77 and FORTRAN/3000. You can perform I/O to the preconnected units $STDIN (unit 5) and $STDLIST (unit 6) in each of the program units. When performing I/O to files, you can use the FORTRAN 77 function FNUM to get a file number; the file number can be passed to the FORTRAN/3000 routine. The FORTRAN/3000 routine can then use the function FSET to assign the file number to a unit number. Also, FORTRAN/3000 interprets all FORTRAN 77 array index types as 1..n regardless of the specified subrange in the FORTRAN 77 source code.

FORTRAN/3000 expects parameters passed by reference. When you pass character parameters to a FORTRAN/3000 program, the FORTRAN language option of the ALIAS compiler directive should be used because FORTRAN/3000 expects the address of the variable when the formal parameter is a character variable. However, FORTRAN 77 normally passes the length of the character variable, as well as the address. If you indicate that the routine being called is a FORTRAN/3000 routine, FORTRAN 77 will not pass the length of the character variable.

Note that logical values may not be consistent between FORTRAN/3000 and FORTRAN 77. Even though the 16-bit value is the same in both the calling and called routines, the compilers check different bits to determine if the value is true or false. FORTRAN/3000 uses the least significant bit, while FORTRAN 77 uses the least significant bit of the first byte.

Example

FORTRAN 77 program:

```
      PROGRAM F77_example
      IMPLICIT NONE
      CHARACTER string*80,charstring*60
      INTEGER*2 filenumber,unit

c  Because a character string is being passed as a parameter, FORTRAN 77
c  must know that FORTRAN/3000 is being called.

$ALIAS fortranprog FORTRAN(%REF,%REF)
      charstring='Pass this character string to a FORTRAN/3000 routine'
      unit=1
      OPEN(unit,FILE='INFILE',STATUS='OLD')          ! Open an input file
                                                     ! and assign to unit 1.
      READ(unit,10)string                            ! Read some data
      WRITE(6,*)string
      filenumber=FNUM(unit)                          ! Get the file number
      CALL fortranprog(filenumber,charstring)        ! Pass the file number
                                                     ! and a character string
                                                     ! to FORTRAN/3000 routine
      READ(unit,10)string                            ! Read some more data.
      WRITE(6,*)string
 10   FORMAT(A80)
      STOP
      END
```

External FORTRAN/3000 subroutine:

```
$CONTROL FILE=2
      SUBROUTINE fortranprog(filenumber,charstring)
      CHARACTER string*80,charstring*60
      INTEGER*2 filenumber,unit,oldfile
c
c  Write the character string that was passed from FORTRAN 77
c
      DISPLAY charstring
      unit=2
c
c  Set up the FORTRAN/3000 FLUT table, assign the filenumber to unit 2
c
      CALL FSET(unit,filenumber,oldfile)
c
c  Read from the file
c
      READ(unit,10)string
      WRITE(6,*)string
 10   FORMAT(S)
      RETURN
      END
```

Calling FORTRAN 77 from FORTRAN/3000

The types of the FORTRAN/3000 actual parameters must match with the FORTRAN 77 formal parameters if you call a FORTRAN 77 routine from a FORTRAN/3000 program. Character variables should not be passed as parameters from a FORTRAN/3000 program to a FORTRAN 77 routine. If you need to use character variables in both the FORTRAN 77 and the FORTRAN/3000 program units, you should use COMMON blocks, as shown in the following example.

The I/O considerations are the same as when calling FORTRAN/3000 from FORTRAN 77/3000; see the previous section "Calling FORTRAN/3000 from FORTRAN 77" for details.

Example

FORTRAN/3000 program:

```
      PROGRAM showcommon
c
c  This example shows the same COMMON used in the FORTRAN 77
c  routine.
c
      COMMON /com/string,intvalue
      CHARACTER*60 string
      INTEGER*4 intvalue
      string='This string will be used in the FORTRAN 77 routine'
      intvalue=40000
      CALL f77showcom
      STOP
      END
```

FORTRAN 77 subroutine:

```
      SUBROUTINE f77showcom
      COMMON /com/string,intvalue
      CHARACTER*60 string
      INTEGER*4 intvalue
      PRINT *,string
      PRINT *,intvalue
      RETURN
      END
```

Pascal/3000

Pascal/3000 is the ANSI Standard Pascal version of Pascal for the HP 3000 computer.

Calling Pascal from FORTRAN 77

A Pascal procedure or function can be called from a FORTRAN 77 program if the data types of the parameters match (see Table A-5). The language code of the ALIAS compiler directive should be used for passing parameters correctly. See the following section and examples for more details.

Table A-5. FORTRAN 77 and PASCAL/3000 Types

FORTRAN 77 Type	Pascal/3000 Type
INTEGER, INTEGER*4	INTEGER Integer subrange beyond the range -32768..32767
INTEGER*2	Integer subrange inside the range -32768..32767
REAL	REAL
REAL*8, DOUBLE PRECISION	LONGREAL
CHARACTER	CHAR
CHARACTER*n	PACKED ARRAY [1..n] OF CHAR
LOGICAL, LOGICAL*4	INTEGER SET (2 words)
LOGICAL*2	Integer subrange inside the range -32768..32767 SET (1 word)
COMPLEX, COMPLEX*8	RECORD real__part : REAL; imag__part : REAL; END;
COMPLEX*16, DOUBLE COMPLEX	RECORD real__part : LONGREAL; imag__part : LONGREAL; END;

FORTRAN 77 cannot pass arrays by value, so you cannot call a Pascal routine with a value parameter of a type corresponding to a FORTRAN 77 array type. For any other type of Pascal value parameter, you must use the %VAL parameter of the ALIAS compiler directive for FORTRAN 77.

All parameters in FORTRAN 77 are word-addressed, except for character variables and character arrays, which are byte-addressed.

All data must be passed through the parameter lists between FORTRAN 77 and Pascal because FORTRAN 77 cannot specify global variables and Pascal cannot specify COMMON blocks. The calling FORTRAN 77 program can have a COMMON area, but the external Pascal procedure or function cannot access COMMON.

Parameter checking should be turned off because Pascal generates different type check values than FORTRAN 77. This can be done by specifying $CHECK__ACTUAL__PARM 0 in the FORTRAN program, or by specifying $CHECK__FORMAL__PARM 0$ in the Pascal procedure.

Example

FORTRAN 77 program:

```
      PROGRAM call_pascal

c  Calling a PASCAL procedure

$ALIAS pasprog PASCAL(%REF)
      CHARACTER str*30
      str='Pass this string'
      CALL pasprog(str)
      PRINT *,str
      END
```

External Pascal/3000 procedure:

```
$SUBPROGRAM$
PROGRAM pascal;
TYPE charstr = PACKED ARRAY[1..30] OF CHAR;

{  Turn parameter checking off because Pascal generates different
   parameter type check values than FORTRAN 77.               }

$CHECK_FORMAL_PARM 0$
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 FORTRAN 77 from Pascal

A FORTRAN 77 subroutine or function can be called from a Pascal program if the data types of the parameters match (see Table A-5). However, be careful when passing character strings and BOOLEAN variables. FORTRAN 77 has a one-word descriptor that describes the maximum length of the string, while PACs (packed array of *char*) in Pascal do not. When a PASCAL character string (PAC) is passed to FORTRAN 77, the string is first expected to be passed by reference (the address of the string) and then the maximum length of the string by value is expected. Also, be careful of BOOLEAN variable differences between the two languages. A byte address is loaded when passing a Pascal BOOLEAN by reference; FORTRAN 77 expects a word address for LOGICAL variables. The following example shows how to pass character strings between Pascal and FORTRAN 77.

Pascal cannot access a FORTRAN 77 COMMON area and cannot pass a file or a label to an external FORTRAN 77 routine.

FORTRAN 77 expects parameters to be passed by reference, with the exception of the maximum length of a character string as mentioned above.

Example

Pascal/3000 program:

```
PROGRAM callfort(OUTPUT);
CONST str_stuff='Pass this string to FORTRAN 77';
TYPE pac = PACKED ARRAY[1..50] OF CHAR;
     SHORTINT = -32768..32767;
VAR str : pac;
    cur_len:shortint;
{  Declare the external FORTRAN 77 program as EXTERNAL SPL
   (because the data types are similar).  We will pass three
   parameters:  the PAC by reference (or the address
   of the string); a one word INTEGER by value specifying the
   maximum length of the PAC; and a one word INTEGER by
   reference, which is the current length of the PAC.        }

PROCEDURE fortprog(VAR str:pac; max_len:SHORTINT;
                   VAR cur_len:shortint);
EXTERNAL SPL;
BEGIN
str:=str_stuff;
{  Get the current length of the PAC              }
cur_len:=strlen(str_stuff);
WRITELN(str);

{  Call the FORTRAN 77 subroutine; pass the PAC, the maximum length
   of the PAC, and the current length of the PAC                    }

fortprog(str,sizeof(str),cur_len);

{  Do a linefeed to print the concatenated string on the following
   line                                                            }
WRITELN;
WRITELN(str);
END.
```

FORTRAN 77 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 FORTRAN 77 uses.
      IMPLICIT NONE
      INTEGER*2 cur_len
c  Use maximum length (the 2nd actual parameter) as the character
c  length.
      CHARACTER str*(*)
c  Concatenate the strings and print result
      str = str(1:cur_len) // ' and then back again'
      PRINT *,str
      RETURN
      END
```

COBOL/3000

The data types of FORTRAN 77 and COBOL greatly differ. Numeric COBOL data types are binary packed-decimal or ASCII format (see Table A-6). By taking the size and the format into consideration, you can successfully match FORTRAN 77 and COBOL types.

Table A-6. COBOL Numeric Types and Formats

<table>
<tr><th>COBOL Type</th><th colspan="3">Format</th></tr>
<tr><td>COMP-3</td><td colspan="3">Packed decimal format with sign in right-most half byte and 2 digits per byte.</td></tr>
<tr><td rowspan="5">COMP</td><td colspan="3">Binary format. Sign bit 0 is +, 1 is -.</td></tr>
<tr><td colspan="2">Size</td><td>Number of Words</td></tr>
<tr><td colspan="2">S9 to S9(4)</td><td>1</td></tr>
<tr><td colspan="2">S9(5) to S9(9)</td><td>2</td></tr>
<tr><td colspan="2">S9(10) to S8(18)</td><td>4</td></tr>
<tr><td rowspan="6">DISPLAY</td><td colspan="3">Unpacked decimal format (ASCII).</td></tr>
<tr><td>Unsigned-</td><td colspan="2">alphanumeric format:
no leading or trailing sign;
1 character per byte.</td></tr>
<tr><td>Sign is -
leading</td><td colspan="2">alphanumeric format;
sign ′overpunched′ in left-most byte.</td></tr>
<tr><td>Sign is -
trailing</td><td colspan="2">alphanumeric format;
sign ′overpunched′ in right-most byte.</td></tr>
<tr><td>Sign is -
leading,
separate</td><td colspan="2">first byte is ASCII ′-′ for negative,
′+′ specifies positive.</td></tr>
<tr><td>Sign is -
trailing,
separate</td><td colspan="2">last byte is ASCII ′-′ for negative,
′+′ specifies positive.</td></tr>
</table>

The following are examples of possible matches between COBOL and FORTRAN 77 types:

COBOL	FORTRAN 77
PIC X(N)	CHARACTER*n
PIC S9(01)-S9(04) COMP	INTEGER*2 {-9999..9999}
PIC S9(05)-S9(09) COMP	INTEGER*4
PIC S9(10)-S9(18) COMP	INTEGER*4 varname(2)

The COBOL types 01 and 77 always start on word boundaries.

The parameter capabilities of COBOL 68 and COBOL II differ. In particular, COBOL 68 cannot pass a parameter on a byte boundary; COBOL II can use the parameter if the @ symbol is specified. COBOL 68 cannot call a FORTRAN function; COBOL II can make the call if the GIVING phrase occurs. FORTRAN 77 cannot accept parameters passed by value. COBOL 68 and COBOL II cannot accept parameters passed by value; reference parameters must be passed to COBOL 68 and COBOL II on word boundaries.

Example

Fortran 77/3000 program:

```
      PROGRAM FORTRAN_COBOL
      IMPLICIT NONE
      INTEGER*4 int1,int2,int3

C  All parameters are passed by reference by default.

      int1 = 25000
      int2 = 30000
      CALL cobprog(int1,int2,int3)
      PRINT *,int3
      END
```

COBOLII/3000 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.
```

USING SYSTEM INTRINSICS

MPE system procedures and functions, which are coded in SPL (Systems Programming Language), are available to you and are called *intrinsics*. These procedures and functions handle individual programming operations and are invoked by a procedure or function call.

There are also FORTRAN 77 intrinsic functions that are part of the ANSI and MIL-STD-1753 standard; these intrinsics are part of the FORTRAN 77 library. The FORTRAN 77 intrinsics do not have to be

defined as intrinsics. Do not confuse system intrinsics with FORTRAN 77 intrinsics; this section only discusses system intrinsics.

For a full description of the system intrinsics, consult the *MPE Intrinsics Reference Manual*; for a full description of the FORTRAN 77 intrinsic functions, see Chapter 5.

Defining System Intrinsics

To define a system intrinsic in a FORTRAN 77 program, the SYSTEM INTRINSIC statement must appear before any executable statement in each procedure or function in which the intrinsic is used. It is not necessary to define how the parameters are passed, because FORTRAN 77 gets this information from the SPLINTR file for system intrinsics.

Example

```
SYSTEM INTRINSIC calendar
```

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 a call is made to an intrinsic, the compiler checks the SPLINTR file to check the actual parameters with the formal parameters. See Table A-7 to match the FORTRAN 77 actual parameters to the intrinsic formal parameters.

Table A-7. SPL and FORTRAN 77 Data Types

SPL TYPE	FORTRAN 77 TYPE
INTEGER	INTEGER*2
DOUBLE	INTEGER*4
REAL	REAL
LONG	DOUBLE PRECISION
BYTE	CHARACTER*1
LOGICAL	LOGICAL*2
INTEGER ARRAY	INTEGER*2 variable(n)
DOUBLE ARRAY	INTEGER*4 variable(n)
BYTE ARRAY	CHARACTER*n
LOGICAL ARRAY	INTEGER*2 variable(n)
BYTE POINTER	Does Not Exist

NOTE: n is an integer representing the size of the ARRAY.

Example

```
   PROGRAM call_intrinsic_print
      IMPLICIT NONE
      CHARACTER*20 message
      INTEGER*2 length,wmessage(10)

c  Associate the character string "message" to the INTEGER array
c  "wmessage" because the first parameter of the PRINT intrinsic is
c  expecting a LOGICAL ARRAY.

      EQUIVALENCE (message,wmessage)
      SYSTEM INTRINSIC print

      length = -len('Print a message')      ! Length of string in bytes
      message = 'Print a message'
      CALL print(wmessage,length,0)
      END
```

In the example above, the first parameter to the PRINT intrinsic is a logical array. See the *MPE Intrinsics Reference Manual* for details. A character string is put in the variable message, which is defined as a CHARACTER*20; CHARACTER*20 has a byte address, but the formal parameter is

expecting a word address (logical array). By using the EQUIVALENCE statement, the character variable `message` is associated with the INTEGER array `wmessage` and the INTEGER array is passed to the intrinsic.

Some of the MPE intrinsics are option variable intrinsics, which means that a partial formal parameter list can be passed to the intrinsic. The MPE intrinsic FOPEN is an option variable intrinsic with up to 13 parameters. You might need to pass only one parameter to the FOPEN intrinsic, like this:

```
CALL FOPEN(,244B)
```

Because the first parameter is not being used, a comma (,) must be put in the parameter list to let the compiler know that the second parameter is the only parameter being passed.

When calling an OPTION VARIABLE intrinsic, you do not have to supply the parameter bit map or dummy values for omitted parameters; FORTRAN 77 supplies this information.

Matching SPL and FORTRAN 77 Data Types

When calling an MPE intrinsic from FORTRAN 77, it does not matter if the intrinsic parameters are reference or value parameters. The parameters are passed as they are expected by the intrinsic.

Table A-7 lists the SPL types that occur in the intrinsic calls and the matching FORTRAN 77 types.

USING HP 3000 SUBSYSTEMS

This section describes how to use HP 3000 subsystems in a FORTRAN 77 program. Examples of FORTRAN 77 programs calling Sort/3000, Image/3000, and VPLUS are included.

Sort-Merge

The Sort-Merge subsystem uses S-relative addressing techniques, which can potentially conflict with the object code generated by the FORTRAN 77 compiler. If the loop counter is type INTEGER*2, the FORTRAN 77 compiler uses S-relative addressing for loops; therefore, correct nesting of loops is necessary when using Sort-Merge. For example, if SORTINIT is put inside a loop and SORTEND is not put at the same level in the loop, errors can occur. An outline of a possible form of a FORTRAN 77 program using the Sort-Merge subsystem is shown below:

```
PROGRAM sort
CALL sortinit(...)
CALL read_file(...)
CALL write_file(...)
CALL sortend
STOP
END

SUBROUTINE read_file(...)
    .
sortinput(...)
    .
RETURN
END

SUBROUTINE write_file(...)
    .
sortoutput(...)
    .
RETURN
END
```

This program uses the outline shown above:

```
$STANDARD_LEVEL SYSTEM
      PROGRAM sort_file
      IMPLICIT NONE
      INTEGER*2 numkeys,keys(3),statistics(12),reclen,spacealloc
      CHARACTER*80 buffer
      SYSTEM INTRINSIC sortinit,sortend,sortstat

c  ****   M A I N   P R O G R A M   ****

c  Call sortinitial to start sort program
      numkeys=1                           ! One field to sort
      keys(1)=1                           ! First character of key to sort
      keys(2)= 20                         ! Sort 20 characters
      keys(3)=0                           ! Sorting character data in
                                          ! ascending order
      reclen=80                           ! 80 BYTE records
      spacealloc=-7000                    ! Leave 7000 words on the stack
                                          ! for FORTRAN 77
c  Initialize the SORT

      CALL sortinit(,,,reclen,,numkeys,keys,,,,statistics,,,spacealloc)
      CALL read_unsorted_file             ! Reads the unsorted file
      CALL write_sorted_file              ! Writes to file, sorted
      CALL sortend                        ! Done sorting
      CALL sortstat(statistics)           ! Write out the sorting statistics
      STOP '- Sort Finished'
      END

c  ****   SUBROUTINE TO READ THE FILE FROM WHICH IS TO BE SORTED   ****

      SUBROUTINE read_unsorted_file
      IMPLICIT NONE
      SYSTEM INTRINSIC sortinput
      CHARACTER*80 buffer
      INTEGER*2 ibuffer(40)

c  Associate the character variable to an integer variable so we
c  can pass the integer variable, which is a word address to SORTINPUT

      EQUIVALENCE (buffer,ibuffer)

c   OPEN the unsorted file and read the records into the sort routine

      OPEN(10,FILE='input',STATUS='OLD',ERR=999)
 1       READ(10,3,END=2)buffer           ! Read until EOF encountered
 3       FORMAT(A80)
         CALL SORTINPUT(ibuffer,80)
        GO TO 1
 2    RETURN
 999  STOP 'Could not OPEN INPUT  open file'
      END
```

```
c  ****   SUBROUTINE TO WRITE THE SORTED RECORDS TO AN OUTPUT FILE  ****

      SUBROUTINE write_sorted_file
      IMPLICIT NONE
      SYSTEM INTRINSIC sortoutput
      INTEGER*2 ibuffer(40),length
      CHARACTER*80 buffer

c  Associate the character variable to an integer variable so we
c  can pass the integer variable (a word address) to SORTOUTPUT

      EQUIVALENCE (buffer,ibuffer)

c   Write the records from the sort routine to the output file

      OPEN(11,FILE='output',STATUS='NEW')
 1    CALL sortoutput(ibuffer,length)
      IF (length .gt. 0) then                  ! Write while there are records
         WRITE(11,4)buffer
 4       FORMAT(A80)
         goto 1
         endif
      RETURN
      END
```

Suppose an unsorted disc file INPUT looks like this:

```
Valley Forge        2750 Monroe Blvd.      Valley Forge, PA  19482
Fort Wayne          3702 Rupp Dr.          Fort Wayne, IN    46815
Long Beach          1501 Hughes Way        Long Beach, CA    90610
Santa Clara         3003 Scott Blvd.       Santa Clara, CA   95050
Dallas              930 E. Campbell Rd.    Richardson, TX    75061
```

After running this program, a new file named OUTPUT is sorted by sales office, and sorting statistics are produced. The file OUTPUT looks like this:

```
Dallas              930 E. Campbell Rd.    Richardson, TX    75061
Fort Wayne          3702 Rupp Dr.          Fort Wayne, IN    46815
Long Beach          1501 Hughes Way        Long Beach, CA    90610
Santa Clara         3003 Scott Blvd.       Santa Clara, CA   95050
Valley Forge        2750 Monroe Blvd.      Valley Forge, PA  19482
```

Image

The program below shows how to use FORTRAN 77 data types that can interface with the IMAGE subsystem.

```
$STANDARD_LEVEL SYSTEM
      PROGRAM FTN77_IMAGE
      IMPLICIT NONE
      COMMON /image_rec/ wcontact_name(10),wcontact_division(10),
     *                   wcontact_number(10)
      INTEGER*2 mode,status(10),answer,wsample_db(5),wsample_pass(4),
     *          wsample_ds(8),wlist(15),wcontact_name,wcontact_division,
     *          wcontact_number
      CHARACTER sample_db*10,sample_pass*8,sample_ds*16,list*30
      EQUIVALENCE (sample_db,wsample_db),(sample_pass,wsample_pass),
     *            (sample_ds,wsample_ds),(list,wlist)
      SYSTEM INTRINSIC DBOPEN

c  ****   M A I N   P R O G R A M   ****

      sample_db='  SAMPLE;'
      sample_pass='EASY;'
      sample_ds='INFO-MASTER;'
      list='NAME,LOCATION,PHONE;'
      mode=3
      CALL DBOPEN(wsample_db,wsample_pass,mode,status)     ! Open database
      IF (status(1) .NE. 0) THEN                           ! and check status
         CALL DBEXPLAIN(status)
          STOP 'DB open error'
      END IF
      answer=0
      DO WHILE (answer .NE. 3)
         CALL list_menu(answer)
         IF (answer .EQ. 1) THEN
            CALL add_record(wsample_db,wsample_ds,wlist)
         ELSE IF (answer .EQ. 2) THEN
            CALL list_record(wsample_db,wsample_ds,status,wlist)
              ELSE IF (answer .EQ. 3) THEN
                 CALL finish_up(wsample_db,wsample_ds)
                   ELSE
                   PRINT *,'INVALID RESPONSE, PLEASE REENTER'
         ENDIF
      END DO
      STOP
      END

c  ****   SUBROUTINE TO LIST THE OPTIONS TO THE USER   ****

      SUBROUTINE list_menu(answer)
      IMPLICIT NONE
      INTEGER*2 answer
      PRINT *,'     INFORMATION FILE OPTIONS'
      PRINT *,'        1) ADD A RECORD       '
      PRINT *,'        2) FIND A PERSON      '
      PRINT *,'        3) FINISHED           '
      PRINT *,'                              '

      PRINT *,'PLEASE ENTER DESIRED OPTION => '
      READ(5,1)answer
```

```
 1    FORMAT(I1)
      RETURN
      END


c  ****   SUBROUTINE TO ADD RECORDS TO THE DATABASE   ****

      SUBROUTINE add_record(wsample_db,wsample_ds,wlist)
      IMPLICIT NONE
      COMMON /image_rec/ wcontact_name(10),wcontact_division(10),
     *                   wcontact_number(10)
      CHARACTER*20 contact_name,contact_division,contact_number
      INTEGER*2 wsample_db(5),wsample_ds(8),wlist(15),status(10),
     *          mode,wcontact_name,wcontact_division,wcontact_number
      EQUIVALENCE (wcontact_name,contact_name),
     *            (wcontact_division,contact_division),
     *            (wcontact_number,contact_number)
      SYSTEM INTRINSIC DBPUT,DBEXPLAIN
      PRINT *,'Enter contact name:  '
      READ(5,1)contact_name
      PRINT *,'Enter contact division:  '
      READ(5,1)contact_division
      PRINT *,'Enter contact phone number:  '
      READ(5,1)contact_number
 1    FORMAT(A20)
      mode=1
      CALL DBPUT(wsample_db,wsample_ds,mode,status,wlist,wcontact_name)
      IF (status(1) .NE. 0) THEN
         PRINT *,'FATAL ERROR'
         CALL DBEXPLAIN(status)
         ENDIF
      RETURN
      END

c  ****   SUBROUTINE TO LIST THE LAST RECORD ENTERED   ***

      SUBROUTINE list_record(wsample_db,wsample_ds,status,wlist)
      IMPLICIT NONE
      COMMON /image_rec/ wcontact_name(10),wcontact_division(10),
     *                  wcontact_number(10)
      CHARACTER contact_name*20,contact_division*20,contact_number*20,
     *          search_item*8,name*20
      INTEGER*2 dummy,mode,wsample_db(5),wsample_ds(8),status(10),
     *          item(4),wcontact_name,wcontact_division,wcontact_number,
     *          wlist(15),wname(10)
      EQUIVALENCE (wcontact_name,contact_name),
     *            (wcontact_division,contact_division),
     *            (wcontact_number,contact_number),(item,search_item),
     *            (wname,name)
      SYSTEM INTRINSIC DBFIND,DBGET,DBEXPLAIN
      search_item='NAME;   '
      mode=7
      PRINT *,'Enter contact person => '
      READ(5,1)name
```

```
 1    FORMAT(A20)
      CALL DBGET(wsample_db,wsample_ds,mode,status,wlist,wcontact_name,
     *           wname)
      IF (status(1) .NE. 0) THEN
         PRINT *,'FATAL ERROR'
         CALL DBEXPLAIN(status)
         ENDIF
      PRINT *,contact_name
      PRINT *,contact_division
      PRINT *,contact_number
      RETURN
      END


c ****   SUBROUTINE TO CLOSE THE DATABASE   ****

      SUBROUTINE finish_up(wsample_db,wsample_ds)
      IMPLICIT NONE
      INTEGER*2 status(10),mode,wsample_db(5),wsample_ds(8)
      SYSTEM INTRINSIC DBCLOSE,DBEXPLAIN
      mode=1
      CALL DBCLOSE(wsample_db,wsample_ds,mode,status)      ! Close database
      IF (status(1) .NE. 0) THEN                           ! and check status
c        PRINT *,'DBCLOSE FAILURE'
         CALL DBEXPLAIN(status)
      ELSE
         PRINT *,'Have a good day!'
      ENDIF
      RETURN
      END
```

VPLUS

VPLUS uses the DL-DB area of the stack to store screen or form information. FORTRAN 77 also uses DL-DB for its file table. To avoid any conflict, a FORTRAN 77 program calling the VPLUS subsystem must use language code 5, as shown in the following example. Code 5 signals VPLUS to call the library procedure GETHEAP, which allocates a region of the DL-DB area for exclusive use by VPLUS. When the formsfile is closed, VPLUS calls the procedure RTNHEAP, which releases the region previously reserved for the subsystem.

In general, you should define VPLUS common areas and buffers on word boundaries. Also, it might be necessary to specify the MAXDATA parameter of the :PREP or :RUN commands to enlarge the DL-DB area.

The following program shows how you can construct FORTRAN 77 data structures suitable for calling VPLUS:

```
$STANDARD_LEVEL SYSTEM
$INIT ON
      PROGRAM F77_V3000
      IMPLICIT NONE

c   Define the VPLUS COMAREA

      COMMON /comarea/ comarea(60)

      INTEGER*2 comarea,cstatus,language,comarealen,usrbuflen,cmode,
     *          lastkey,numerrs,windowenh,multiusage,labeloptions,
     *          repeatapp,freezapp,cfnumlines,dbuflen,skip2,lookahead,
     *          deleteflag,showcontrol,skip4,printfilnum,filerrnum,
     *          errfilenum,forms,skip6,skip7,skip8,skip9(2),term_filen,
     *          skip10(5),retries,term_options,environ,usertime,
     *          labelinfo,formstrsize,identifier

      INTEGER*4 numrecs,recnum

      CHARACTER cfname*16,nfname*16,terminal*8,formfile*8

      EQUIVALENCE (comarea(1),cstatus),
     *            (comarea(2),language),
     *            (comarea(3),comarealen),
     *            (comarea(4),usrbuflen),
     *            (comarea(5),cmode),
     *            (comarea(6),lastkey),
     *            (comarea(7),numerrs),
     *            (comarea(8),windowenh),
     *            (comarea(9),multiusage),
     *            (comarea(9),labeloptions),
     *            (comarea(10),cfname),
     *            (comarea(18),nfname),
     *            (comarea(26),repeatapp),
     *            (comarea(27),freezapp),
     *            (comarea(28),cfnumlines),
     *            (comarea(29),dbuflen),
     *            (comarea(30),skip2),
     *            (comarea(31),lookahead),
     *            (comarea(32),deleteflag),
     *            (comarea(33),showcontrol)
      EQUIVALENCE (comarea(34),skip4),
     *            (comarea(35),printfilnum),
     *            (comarea(36),filerrnum),
     *            (comarea(37),errfilenum),
     *            (comarea(38),formstrsize),
     *            (comarea(39),skip6),
     *            (comarea(40),skip7),
     *            (comarea(41),skip8),
     *            (comarea(42),numrecs),
     *            (comarea(44),recnum),
     *            (comarea(46),skip9),
     *            (comarea(48),term_filen),
     *            (comarea(49),skip10),
```

```
     *              (comarea(54),retries),
     *              (comarea(55),term_options),
     *              (comarea(56),environ),
     *              (comarea(57),usertime),
     *              (comarea(58),identifier),
     *              (comarea(59),labelinfo)

      SYSTEM INTRINSIC vopenterm,vopenformf,vcloseterm,vcloseformf

      language=5                                     ! "PASCAL", (uses DL-DB)
      comarealen=60
      terminal='x '
      formfile='form '
      CALL vopenterm(comarea,terminal)                  ! Open the terminal
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR ',cstatus,' FSerr ',filerrnum
         CALL verr
         ENDIF

      CALL vopenformf(comarea,formfile)                 ! Open forms file
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR ',cstatus,' FSerr ',filerrnum
         CALL verr
         ENDIF

      CALL vgetnextform(comarea)                        ! Get a form
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR ',cstatus
         CALL verr
         ENDIF

      CALL vshowform(comarea)                           ! Show the form
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR ',cstatus
         CALL verr
         ENDIF

      CALL vcloseformf(comarea)                         ! Close the form
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR'
         CALL verr
         ENDIF

      CALL vcloseterm(comarea)                          ! Close the terminal
      IF (cstatus .NE. 0) THEN                          ! and check status
         PRINT *,'FATAL ERROR ',cstatus
         CALL verr
         ENDIF

      STOP
      END

c  Error handling routine to print error message
```

```
      SUBROUTINE verr
      IMPLICIT NONE
      COMMON /comarea/ comarea(60)
      INTEGER*2 comarea,msglen,i
      CHARACTER*1 buffer(80)
      SYSTEM INTRINSIC verrmsg
      CALL verrmsg(comarea,buffer,80,msglen)
      PRINT *,(buffer(i),i=1,msglen)
      RETURN
      END

c  Used to initialize the common block - COMAREA

      BLOCK DATA clear
      COMMON /comarea/ comarea(60)
      INTEGER*2 comarea
      DATA comarea/60*0/
      END
```

STACK ARCHITECTURE

This section gives a brief overview of the code segment and data stack layouts.

Figure A-1 shows the layout of a code segment. If the subroutine, function, or program is preceded by a $COPYRIGHT statement, the copyright message will precede that program unit in the code segment.

Following the copyright statement are the format statements and their lengths, the program code, and the code constants (parameter statements and internal compiler constants). This information continues to be placed in the segment for each of the procedures and functions in the opposite order in which they were compiled. The last program unit compiled will be the first in the code segment. Finally, the end of the segment contains the Segment Transfer Table (STT).

Figure A-2 shows the layout of the data stack. FORTRAN 77 uses part of DL-DB for its file list (I/O Library Globals). DB-12 contains a pointer to the beginning of the I/O Library Globals. In Primary DB, pointers to COMMON start at DB+0. Secondary DB contains the storage for DATA and SAVE variables and arrays, COMMON variables and arrays, and the PARM and INFO string length and pointer at Q-6 through Q-4. Following the PARM and INFO string is the initial stack marker. Beginning at Q_{i+1} are pointers and storage for the main program variables and arrays. If the program calls a subroutine or function, there is another stack marker and, beginning at Q_{i+1}, pointers and storage for the variables and arrays local to that subroutine or function. The remainder of the procedure and function variables are stored in the same way.

Fig. A-1. Sample Code Segment Layout for FORTRAN 77

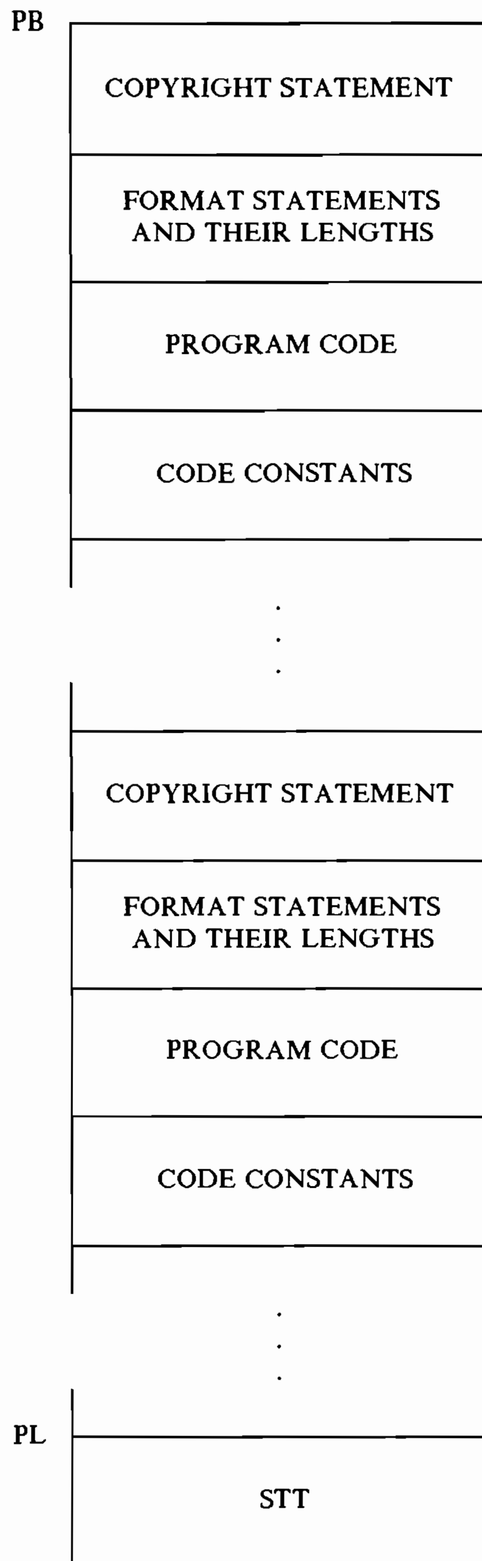
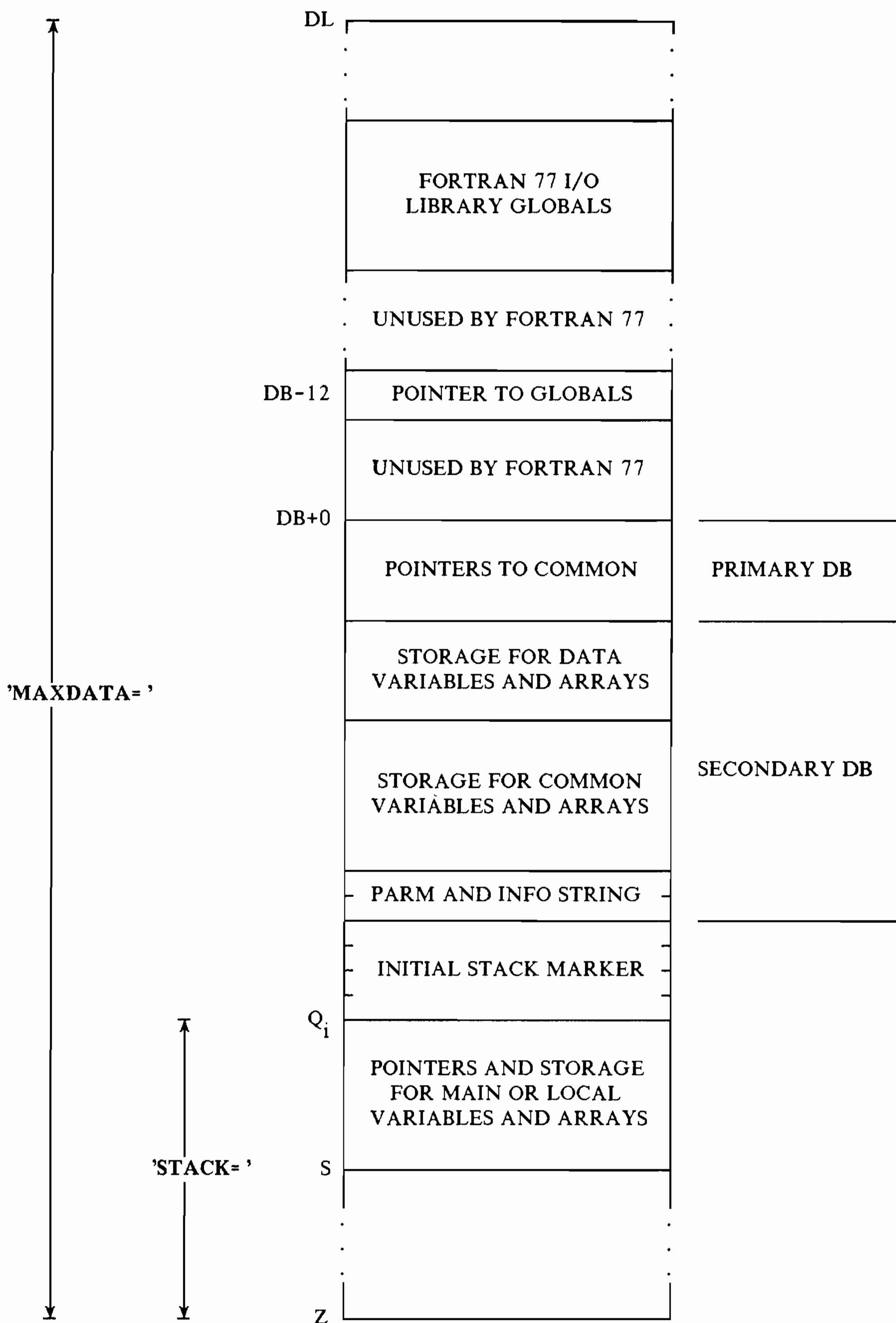


Fig. A-2. Sample Data Stack Layout for FORTRAN 77



INDEX

/ edit descriptor 2–24
$ edit descriptor 2–25
' edit descriptor 2–29
: edit descriptor 2–35
@ format descriptor 2–8

A

A format descriptor 2–15
ABS intrinsic function 5–7
ACCESS='DIRECT' option A–3
ACCESS='SEQUENTIAL' option A–3
ACCESS='SEQUENTIAL' option 3–2
accessing the representation of logical values 7–4
ACOS intrinsic function 5–8
ACOSH intrinsic function 5–8
actual arguments
  in subprograms 4–15
  in subroutines 4–3
addressing mode 1–1
adjustable dimensions 4–22
AIMAG intrinsic function 5–21
AINT intrinsic function 5–9
alignment of data 1–1
ALOG intrinsic function 5–27
ALOG10 intrinsic function 5–27
alternate returns from a subroutine 4–4
AMAX0 intrinsic function 5–28
AMAX1 intrinsic function 5–28
AMIN0 intrinsic function 5–28
AMIN1 intrinsic function 5–28
AMOD intrinsic function 5–29
ANINT intrinsic function 5–9
ANSI compiler directive 7–1
apostrophes for character data 2–2
appending to a file 3–9
argument lists 4–15
arguments to subprograms 4–15
arrays 2–40
arrays
  adjustable dimensions 4–22
  assumed-size 4–24
  passed in a subprogram 4–21
ASIN intrinsic function 5–10
ASINH intrinsic function 5–10
ASSIGN statement 2–39
assumed-size arrays 4–24
ATAN intrinsic function 5–11
ATAN2 intrinsic function 5–12
ATANH intrinsic function 5–11

B

BACKSPACE statement 3–20
blank common blocks 4–28
BLANK specifier 3–12
blanks in input field
  BN descriptor 2–37
  BZ descriptor 2–38
BLOCK DATA statement 4–31
block data subprograms 4–31
BN edit descriptor 2–37
BTEST intrinsic function 5–12
BZ edit descriptor 2–38

C

CABS intrinsic function 5–7
CALL statement
  alternate return 4–4
  invoking a subroutine 4–3

carriage control files
  :FILE equation A-5
  disc files A-5
  terminals and line printers A-5
CCOS intrinsic function 5-14
CEXP intrinsic function 5-17
CHAR intrinsic function 5-13
character data in a subprogram 4-19
character data
  list-directed 2-2
character format descriptors
  A descriptor 2-15
  input field 2-17
  output field 2-18
  R descriptor 2-15
character positions
  T edit descriptor 2-27
  TL edit descriptor 2-28
  TR edit descriptor 2-29
  X edit descriptor 2-25
character variables
  EQUIVALENCE statement 1-5
CLOG intrinsic function 5-27
CLOSE statement
  description 3-1
  STATUS specifier 3-3, A-4
CMPLX intrinsic function 5-13
COBOL/3000 A-32
code segment A-46
code space efficiency 6-7
$CODE_OFFSETS compiler directive A-12
colon edit descriptor 2-35
comments A-20
Comments 7-7
Common blocks 7-3
common blocks
  blank common 4-28
  description 4-28
  EQUIVALENCE statement 1-6
  labeled common 4-30
COMMON statement 4-28
compilation directives A-20
compile time efficiency 6-1
complex format descriptors 2-12
CONJG intrinsic function 5-14
connecting files 3-1
constants passed in a subprogram 4-17
COS intrinsic function 5-14
COSH intrinsic function 5-15
creating a new file 3-4
CSIN intrinsic function 5-32
CSQRT intrinsic function 5-34
CTAN intrinsic function 5-34
current record 3-1

D

D format descriptor 2-12
DABS intrinsic function 5-7
DACOS intrinsic function 5-8
DACOSH intrinsic function 5-8
DASIN intrinsic function 5-10
DASINH intrinsic function 5-10
data alignment 1-1, 1-7
data classes 1-2
data space efficiency 6-8
data stack A-46
data storage 1-1
data storage, consistent 7-2
DATAN intrinsic function 5-11
DATAN2 intrinsic function 5-12
DATANH intrinsic function 5-11
DBLE intrinsic function 5-15
DCMPLX intrinsic function 5-16
DCONJG intrinsic function 5-14
DCOS intrinsic function 5-14
DCOSH intrinsic function 5-15
DDIM intrinsic function 5-16
DDINT intrinsic function 5-9
debugging programs A-12
default file properties 3-1
descriptor mode of addressing 1-1, A-12
DEXP intrinsic function 5-17
DFLOAT intrinsic function 5-15
DIM intrinsic function 5-16
DIMAG intrinsic function 5-21
dimensions, adjustable 4-22
DINT intrinsic function 5-9
direct access file
  creating 3-13
direct access files
  reading and writing 3-12
direct mode of addressing 1-1, A-12
disc files 3-1, A-5
disconnecting files 3-1
DLOG intrinsic function 5-27
DLOG10 intrinsic function 5-27
DMAX1 intrinsic function 5-28

DMIN1 intrinsic function 5–28
DMOD intrinsic function 5–29
DNINT intrinsic function 5–9
DO loop, implied 2–40
double complex format descriptors 2–12
double precision format descriptors 2–12
DPROD intrinsic function 5–16
DSIGN intrinsic function 5–32
DSIN intrinsic function 5–32
DSINH intrinsic function 5–33
DSQRT intrinsic function 5–34
DTAN intrinsic function 5–34
DTANH intrinsic function 5–35
dummy arguments
  in subprograms 4–15
  in subroutines 4–3

E

E format descriptor 2–12
edit descriptors 2–7
edit descriptors
  table 2–8
efficiency
  code space 6–7, A–19
  compile time 6–1, A–18
  data space 6–8, A–19
  run-time 6–2, A–18
efficient programs 6–1
END statement
  subroutines 4–3
end-of-file record 3–11
ENDFILE statement 3–20
entry points into a subprogram 4–25
ENTRY statement 4–25
EQUIVALENCE statement
  array elements 1–3
  arrays with different dimensions 1–4
  avoid using 7–3
  character variables 1–5
  common blocks 1–6
  data alignment 1–7
  data storage 1–2
ERR specifier 3–4
errors, file handling 3–2
examples
  file handling 3–22
  using file positioning statements 3–20
EXP intrinsic function 5–17
expressions passed in a subprogram 4–18
EXTERNAL statement 5–1

F

F format descriptor 2–12
file access
  direct 3–12
  sequential 3–11
:FILE equation A–5
file handling errors
  ERR specifier 3–4
  IOSTAT specifier 3–4
  STATUS specifier 3–2
file operations on the HP 3000 A–1
file pointer 3–1
file pointer, postioning 3–20
file positioning
  BACKSPACE statement 3–20
  ENDFILE statement 3–20
  examples 3–22
  REWIND statement 3–20
files
  creating 3–4
  direct 3–12
  formatted 3–16
  internal 3–24
  reading 3–7
  sequential 3–11
  unformatted 3–16
FLOAT intrinsic function 5–31
FNUM procedure A–8
FOPEN intrinsic A–1
FORM='FORMATTED' option A–3
FORM='UNFORMATTED' option A–3
FORM='FORMATTED' option 3–2
format descriptors 2–7
format descriptors
  table 2–7
format specifications 2–8
FORMAT statement 2–4
formatted files 3–16
formatted I/O 2–1
formatted statements 2–4
formatted statements
  formatted input 2–4
  formatted output 2–6
FORTRAN/3000 A–25
FSET procedure A–6

FTN05 A-2
FTN06 A-2
functions
  description 4-7
  statement 4-13
  subprogram 4-7

G

G format descriptor 2-12
generic function names 5-1

H

H edit descriptor 2-30
HABS intrinsic function 5-7
HBCLR intrinsic function 5-18
HBITS intrinsic function 5-18
HBSET intrinsic function 5-19
HDIM intrinsic function 5-16
HIAND intrinsic function 5-18
HIEOR intrinsic function 5-20, 5-24
HIOR intrinsic function 5-22
HMOD intrinsic function 5-29
HMVBITS intrinsic function 5-29
HNOT intrinsic function 5-31
HP 3000 A-1
HSHFT intrinsic function 5-23
HSHFTC intrinsic function 5-23
HSIGN intrinsic function 5-32
HTEST intrinsic function 5-12

I

I format descriptor 2-8
IABS intrinsic function 5-7
IAND intrinsic function 5-18
IBCLR intrinsic function 5-18
IBITS intrinsic function 5-18
IBSET intrinsic function 5-19
ICHAR intrinsic function 5-19
IDIM intrinsic function 5-16
IDINT intrinsic function 5-22
IDNINT intrinsic function 5-31
IEOR intrinsic function 5-20, 5-24
IFIX intrinsic function 5-22
IMAG intrinsic function 5-21
IMAGE subsystem A-40
IMPLICIT statement 5-1
implied DO loop 2-40
INDEX intrinsic function 5-21
indirect mode of addressing 1-1, A-12
Initializing data 7-4
input
  unformatted 3-15
INQUIRE statement 3-18
INT intrinsic function 5-22
integer format descriptors
  I descriptor 2-8
  input field 2-9
  K descriptor 2-8
  O descriptor 2-8
  output field 2-11
  Z descriptor 2-8
  @ descriptor 2-8
interfacing with other languages
  COBOL/3000 A-32
  FORTRAN/3000 A-25
  Pascal/3000 A-28
  SPL/3000 A-22
internal files
  description 3-24
  reading 3-24
  writing 3-26
intrinsic functions
  description 5-1
  descriptions 5-7
  generic names 5-1
  invoking 5-1
  specific names 5-1
  summary 5-2
invoking an intrinsic function 5-1
IOR intrinsic function 5-22
IOSTAT specifier 3-4
ISHFT intrinsic function 5-23
ISHFTC intrinsic function 5-23
ISIGN intrinsic function 5-32
IXOR intrinsic function 5-20, 5-24

K

K format descriptor 2-8

L

L format descriptor 2-20
labeled common blocks 4-30
LEN intrinsic function 5-24
length specifications 7-3
LGE intrinsic function 5-25

LGT intrinsic function 5–25
list directed I/O 2–1
list directed input 2–1
list directed output 2–2
literal data
  H edit descriptor 2–30
  ' edit descriptor 2–29
LLE intrinsic function 5–26
LLT intrinsic function 5–26
LOG intrinsic function 5–27
LOG10 intrinsic function 5–27
logical format descriptor
  input field 2–21
  L descriptor 2–20
LONG compiler directive 7–2

M

magnetic tapes A–5
maintaining parameter type and length consistency 7–4
MAX intrinsic function 5–28
MAX0 intrinsic function 5–28
MAX1 intrinsic function 5–28
MIN intrinsic function 5–28
MIN0 intrinsic function 5–28
MIN1 intrinsic function 5–28
MOD intrinsic function 5–29
modifiable programs 7–5
MPE debug A–12
MPE operating system A–1
multiple entries into subprograms 4–25
MVBITS intrinsic function 5–29

N

new files 3–4
new lines 2–24
NINT intrinsic function 5–31
NL edit descriptor 2–25
NN edit descriptor 2–25
NOT intrinsic function 5–31
number of bytes
  Q edit descriptor 2–35

O

O format descriptor 2–8
OPEN statement A–1
OPEN statement
  ACCESS='SEQUENTIAL' option 3–2
  appending to a file 3–9
  connecting a file 3–1
  creating a file A–2
  ERR specifier 3–4
  FORM='FORMATTED' option 3–2
  IOSTAT specifier 3–4
  reporting errors 3–2
  STATUS specifier 3–2
  STATUS='NEW' option 3–4
  STATUS='OLD' option 3–7
  STATUS='UNKNOWN' option 3–2
optimization techniques 6–1
output
  unformatted 3–16

P

P edit descriptor 2–31
PARAMETER statement 2–39, 7–8
parameter types 7–4
Pascal/3000 A–28
passing arrays 4–21
passing character data 4–19
passing constants 4–17
passing expressions 4–18
passing subprograms 4–25
plus signs
  S edit descriptor 2–34
  SP edit descriptor 2–34
  SS edit descriptor 2–34
portable programs 7–1, A–19
preconnected unit numbers 2–1
predefined units
  FTN05 A–2
  FTN06 A–2
  $STDINX A–1
  $STDLIST A–1
processing new lines
  / descriptor 2–24
  NL descriptor 2–25
  NN descriptor 2–25
  $ descriptor 2–25
program unit 4–1
programming for portability 7–1

Q

Q edit descriptor 2–35

R

R format descriptor 2-15
$RANGE compiler directive A-17
READ statement
  formatted input 2-4
  list directed input 2-1
  unformatted 3-15
reading an existing file 3-7
real format descriptors
  D descriptor 2-12
  E descriptor 2-12
  F descriptor 2-12
  G descriptor 2-12
  input field 2-12
  output field 2-13
REAL intrinsic function 5-31
recursive subroutines 4-2
repeating specifications 2-22
reporting file handling errors 3-2
restricting programs to HP FORTRAN 77 standard 7-1
RETURN statement
  alternate return 4-4
  containing an expression 4-5
  subroutines 4-3
REWIND statement 3-20
RUN command parameters A-11
run-time efficiency 6-2

S

S edit descriptor 2-34
SAVE statement 4-33
scale factors, P edit descriptor 2-31
segment transfer table A-46
sequential access files 3-11
SHORT compiler directive 7-2
SIGN intrinsic function 5-32
SIN intrinsic function 5-32
SINH intrinsic function 5-33
slash in input field 2-2
SNGL intrinsic function 5-31
Sort-Merge subsystem A-37
SP edit descriptor 2-34
specific function names 5-1
SPL/3000 A-22
SQRT intrinsic function 5-34
SS edit descriptor 2-34
stack architecture A-46
standard features of HP FORTRAN 77 7-1
standard input 2-1
standard output 2-1
STATUS specifier
  'NEW' status A-2
  'OLD' status A-2
  'SCRATCH' status A-3
  'UNKNOWN' status A-3
  'DELETE' status 3-3
  'KEEP' status 3-3
  'NEW' status 3-2
  'OLD' status 3-2
  'SCRATCH' status 3-2
  'UNKNOWN' status 3-2
STATUS='NEW' option 3-4
STATUS='OLD' option 3-7
STATUS='UNKNOWN' option 3-2
$STDINX A-1
$STDLIST A-1
STOP statement
  subroutines 4-3
storage allocation
  addressing mode 1-1
  EQUIVALENCE statement 1-2
  variable types 1-1
subprograms
  arguments 4-15
  block data 4-31
  categories 4-1
  description 1-1
  in other languages 4-34
  multiple entries 4-25
  passing 4-25
SUBROUTINE statement 4-2
subroutines
  alternate returns 4-4
  description 4-2
  invoking 4-3
  recursive 4-2
  structure 4-2
subsystems
  IMAGE A-40
  Sort-Merge A-37
  VPLUS A-43
system intrinsics
  defining A-34
  description A-34
  matching data types A-37
  matching parameters A-35

T

T edit descriptor 2-27
$TABLES compiler directive A-12
TAN intrinsic function 5-34
TANH intrinsic function 5-35
terminals and line printers A-5
TL edit descriptor 2-28
TR edit descriptor 2-29
transportable programs 7-1

U

unformatted files 3-16
unformatted input 3-15
unformatted output 3-16
unformatted READ statement 3-15
unformatted WRITE statement 3-16
unit numbers 2-1
UNITCONTROL procedure A-8
unstructured features 7-10
using consistent data storage 7-2

V

variable types 1-1
VPLUS subsystem A-43

W

WRITE statement
  formatted 3-17
  formatted output 2-6
  list directed output 2-2
  unformatted 3-16, 3-17
writing efficient programs 6-1

X

X edit descriptor 2-25

Z

Z format descriptor 2-8
ZABS intrinsic function 5-7
ZCOS intrinsic function 5-14
ZEXP intrinsic function 5-17
ZLOG intrinsic function 5-27
ZSIN intrinsic function 5-32
ZSQRT intrinsic function 5-34
ZTAN intrinsic function 5-34