FORTRAN Reference Manual

Abstract

This reference manual documents the HP implementation of the FORTRAN 77 language including HP extensions. Readers should already be familiar with the FORTRAN 77 language.

Product Version

FORTRAN D20

Supported Release Version Updates (RVUs)

This manual supports D20.00 and all subsequent D-series RVUs, and G02.00 and all subsequent G-series RVUs until otherwise indicated in a new edition.

Part Number	Published
528615-001	August 2004

Document History

Part Number	Product Version	Published
82515 A00	FORTRAN B40	October 1986
15546	FORTRAN C20	March 1989
065115	FORTRAN D10	January 1993
528615-001	FORTRAN D20	August 2004

FORTRAN Reference Manual

2. Language Elements (continued)

3. Expressions

4. Program Units

4. Program Units (continued)

5. Introduction to File I/O in the HP NonStop Environment

6. Introduction to Statements

7. Statements

7. Statements (continued)

8. Intrinsic Functions

8. Intrinsic Functions (continued)

9. Program Compilation

10. Compiler Directives

10. Compiler Directives (continued)

10. Compiler Directives (continued)

11. Running and Debugging Programs

12. Memory Organization

13. Mixed-Language Programming

13. Mixed-Language Programming (continued)

14. Interprocess Communication

15. Utility Routines

15. Utility Routines (continued)

16. Fault-Tolerant Programming

A. ASCII Character Set

B. Syntax Summary

C. Converting Programs to HP FORTRAN

D. Data Type Correspondence and Return Value Sizes

E. Compiler Limits

F. Compile-Time Diagnostic Messages

G. Run-Time Diagnostic Messages

H. Hollerith Constants and Punch Card Codes

Glossary

Index

Examples

Figures

Tables

Tables (continued)

Tables (continued)

What's New in This Manual

Manual Information

Abstract

This reference manual documents the HP implementation of the FORTRAN 77 language including HP extensions. Readers should already be familiar with the FORTRAN 77 language.

Product Version

FORTRAN D20

Supported Release Version Updates (RVUs)

This manual supports D20.00 and all subsequent D-series RVUs, and G02.00 and all subsequent G-series RVUs until otherwise indicated in a new edition.

Part Number	Published
528615-001	August 2004

Document History

Part Number	Product Version	Published
82515 A00	FORTRAN B40	October 1986
15546	FORTRAN C20	March 1989
065115	FORTRAN D10	January 1993
528615-001	FORTRAN D20	August 2004

New and Changed Information

This publication has been updated to reflect new product names:

- Since product names are changing over time, this publication might contain both HP and Compaq product names.
- Product names in graphic representations are consistent with the current product interface.

The following changes have been made for this edition:

- Added message 255 on page G-8.
- Migrated the manual to current authoring tools.
- Rebranded and updated for new release terminology and manual titles.

About This Manual

This manual describes the HP implementation of the full ANSI FORTRAN 77 (X3.9-1978) language. It also describes HP extensions that optimize FORTRAN for the transaction-oriented, fault-tolerant environment. All HP extensions provide capabilities beyond those stated in the FORTRAN standard.

Who Should Use This Manual

This manual is for system and application programmers.

Manual Organization

Table i. Summary of Contents (page 1 of 2)

Section	Title	This section . . .
1	Summary of HP Extensions	An introduction to FORTRAN. If you are thoroughly familiar with FORTRAN, you need read only the description of RECORD structures in Section 2, Language Elements and in Section 5, Introduction to File I/O in the HP NonStop Environment.
2	Language Elements	
3	Expressions	
4	Program Units	
5	Introduction to File I/O in the HP NonStop Environment	
6	Introduction to Statements	
7	Statements	Language statements and intrinsic functions.
8	Intrinsic Functions	
9	Intrinsic Functions	Using compiler directives and compiling and running FORTRAN programs.
10	Compiler Directives	
11	Running and Debugging Programs	
12	Memory Organization	Using features of the HP FORTRAN environment.
13	Mixed-Language Programming	
14	Interprocess Communication	
15	Utility Routines	
16	Fault-Tolerant Programming	
A	ASCII Character Set	Lists ASCII characters.
B	Syntax Summary	Provides a syntax summary for FORTRAN statements and compiler directives.
C	Converting Programs to HP FORTRAN	Contains suggestions to convert a FORTRAN application program that was not written for HP FORTRAN to the syntax and semantics of HP FORTRAN.

Table i. Summary of Contents (page 2 of 2)

Section	Title	This section . . .
D	Data Type Correspondence and Return Value Sizes	Lists the return value size generated by HP NonStop language compilers for each data type.
E	Compiler Limits	Summarizes the limits of the FORTRAN compiler
F	Compile-Time Diagnostic Messages	Lists the FORTRAN 77 compiler diagnostic messages that FORTRAN might report in the program listing.
G	Run-Time Diagnostic Messages	Describes the features of FORTRAN statements and utility routines. Also, describes the format and content of the diagnostic messages.
H	Hollerith Constants and Punch Card Codes	Describes Hollerith constants and punch card codes.

The Glossary provides definitions for terms used in the manual.

Although Section 7, Statements, Section 8, Intrinsic Functions, and Section 10, Compiler Directives, comprise the reference part of the manual, consult the other sections for further clarification and more complete examples.

Prerequisites

If you are unfamiliar with the FORTRAN programming language, any of the numerous tutorial texts currently available are recommended as supplemental reading. If you are unfamiliar with the TNS systems or the HP NonStop™ Kernel operating system, refer to the following HP manuals:

Manual	Description
Introduction to Tandem NonStop Systems	Describes the application environment, architecture, and networking capabilities of NonStop systems and explains basic concepts, terms, and entities in the NonStop environment.
Introduction to D-Series Systems	Provides an overview of D-series enhancements to the HP NonStop Kernel operating system.
Guardian Application Conversion Guide	Describes how to convert C, COBOL85, Pascal, TAL, and TACL applications to use the extended features of the NonStop Kernel.
Guardian User's Guide	Explains how to run program files and how to create keyed files using the File Utility Program (FUP).
Guardian Procedure Calls Reference Manual	Describes the syntax for Guardian procedure calls.
ENSCRIBE Programmer's Guide	Describes the Enscribe database record manager.

Manual	Description
Spooler Programmer's Guide	Describes the HP spooler program.
CRE Programmer's Guide	Explains features of the CRE, including file sharing and error handling.
TACL Reference Manual	Describes command interpreter commands.
Inspect Manual	Describes how to use Inspect, an interactive symbolic debugging utility.
Binder Manual	Explains the binding of object files.
Debug Manual	Describes how to use Debug, an interactive low-level debugging utility.
CROSSREF Manual	Explains how to use the cross-reference listing tool.

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

> This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

Event number = *number* [Subject = *first-subject-value*]

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

proc-name trapped [in SQL | in SQL file system]

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

obj-type obj-name state changed to *state*, caused by
{ Object | Operator | Service }

process-name State changed from *old-objstate* to *objstate*
{ Operator Request. }
{ Unknown.   }

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400

%B101111

%H2F

P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME       token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER       token-type ZSPI-TYP-FNAME32.         !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Summary of HP Extensions

HP FORTRAN for NonStop systems implements the full ANSI FORTRAN 77 (X3.9-1978) language. In addition, HP FORTRAN extensions to the ANSI standard enable you to:

- Use 31-character identifier names.
- Use RECORD declarations to define and reference data structures containing data of different types.
- Combine procedures written in C, COBOL85, FORTRAN, Pascal, and TAL into an executable program.
- Write fault-tolerant programs.
- Access key-sequenced, relative, and entry-sequenced files by primary or alternate keys, and make full use of the ENSCRIBE facilities.
- Use standard READ and WRITE statements to communicate with other processes.
- Invoke Guardian system procedures as if they were FORTRAN subroutines.
- Use the Binder, Crossref, and Inspect development tools.
- Access extended data storage and use large common and local data blocks.

These features are described briefly in these topics:

Character Set and Identifier Names

The HP FORTRAN character set includes the following symbols that are not part of the ANSI standard character set:

- _ for use in names
- % to designate octal integers
- ^ for use in RECORD references and section names
- ? to identify a line of source that contains compiler directives
- \ for pass-by-value parameters
- " for defining the string that the PAGE directive prints at the top of each page

For additional information, see Section 2, Language Elements.

The names of external subprograms, common block names, variables, arrays, and statement functions can be up to 31 characters long.

Data Types

You can declare word, doubleword, or quadrupleword INTEGER entities and word or doubleword LOGICAL entities. For additional information, see Section 2, Language Elements.

You can use Hollerith constants in DATA statements or as actual arguments in CALL statements. See Section H, Hollerith Constants and Punch Card Codes.

Using the RECORD and END RECORD statements, you can declare RECORD structures that mix different data types in one record and support the processing of database records. For additional information, see Section 2, Language Elements.

Procedures

HP FORTRAN supports recursive subprograms. For additional information, see Section 4, Program Units.

Input and Output Operations

You can mix numeric data for real and integer entities: on input, numeric data is automatically converted to the data type of the corresponding input list item; on output, numeric data is converted to the data type required by the format specification. For additional information, see Section 5, Introduction to File I/O in the HP NonStop Environment.

A record can be shorter than the entity specified in the I/O list. The formatter fills the remainder of a short input record with blanks. For additional information, see Section 5, Introduction to File I/O in the HP NonStop Environment.

FORMAT statements can specify binary, octal, and hexadecimal numeric conversion. For additional information, see Section 7, Statements.

Files

Your program can use both formatted and unformatted I/O on the same file.

You can position a structured file for keyed access by using the POSITION statement. You can update and delete records in structured files by using the UPDATE specifier with READ and WRITE statements. For additional information, see Section 5, Introduction to File I/O in the HP NonStop Environment, and Section 7, Statements.

Your program can open a file multiple times by having more than one UNIT reference the file. For additional information, see Section 5, Introduction to File I/O in the HP NonStop Environment.

The function FILENUM returns the Guardian file number associated with the specified unit. For additional information, see Section 8, Intrinsic Functions.

The TIMEOUT specifier for the READ and WRITE statements provides timed access to files. For additional information, see the description of the READ and WRITE statements in Section 7, Statements.

The LENGTH specifier for the READ statement obtains the actual length of a variablelength record. The LENGTH specifier for the WRITE statement determines the last character position used in a data transfer to an internal file. For additional information, see the description of the READ and WRITE statements inSection 7, Statements.

The PROMPT and PROMPTLENGTH specifiers for the READ statement provide prompting for input from interactive terminals. The OPEN statement's control list includes the optional SYNCDEPTH, PROTECT, MODE, SPACECONTROL, STACK, and TIMED specifiers for additional file control capabilities. For more information, see the descriptions of the READ and OPEN statements in Section 7, Statements.

Access to Operating System Procedures

Using the GUARDIAN directive described in Section 10, Compiler Directives, FORTRAN programs can invoke Guardian procedures as subroutines or external functions as if they were written in FORTRAN. For additional information, see Section 13, Mixed-Language Programming, and Section 15, Utility Routines.

Mixed-Language Programming

You can specify external functions and subroutines in C, COBOL85, Pascal, or TAL and bind these modules, along with your FORTRAN modules, into a single executable program using the Binder program.

For additional information, see Section 9, Program Compilation, and Section 13, Mixed-Language Programming.

Memory Management

A FORTRAN program can have up to 32 code segments—each code segment containing up to 128KB—and up to 127.5MB of data storage in an extended data segment.

You can use compiler directives to:

- Allocate space in upper memory and in extended memory
- Reference items in common blocks using indexed addressing

For additional information, see Section 12, Memory Organization.

Fault-Tolerant Programming

The START BACKUP and CHECKPOINT statements support fault-tolerant processes. For additional information, see Section 6, Introduction to Statements.

Interprocess Communication

The $RECEIVE facility is available to FORTRAN programs for interprocess communication.

- The RECEIVE directive specifies $RECEIVE message-handling options.
- The SOURCE specifier with the READ statement receives message information.
- The MSGNUM and REPLY specifiers with the WRITE statement enable you to reply to messages received from $RECEIVE.

For additional information, see Section 14, Interprocess Communication.

2 Language Elements

Topics covered in this section include:

The FORTRAN Character Set

The FORTRAN character set, shown in Table 2-1 on page 2-2, consists of 26 letters, 10 digits, and 19 special characters. These characters are the only acceptable characters in the text of a FORTRAN statement, although you can use any character in the ASCII character set in character constants, literals, comment lines, and Hollerith field descriptors. The FORTRAN collating sequence follows the order of the ASCII character set, shown in Appendix A, ASCII Character Set.

The FORTRAN compiler ignores case except in character constants. The following two statements are identical in FORTRAN:

```
PROGRAM random number generator
PROGRAM RANDOM NUMBER GENERATOR
```

To help you distinguish keywords from symbols in this manual, keywords are shown in uppercase characters, and symbolic names are shown in lowercase characters.

Table 2-1. FORTRAN Character Set

Alphabetic	a b c d e f g h i j k l m n o p q r s t u v w x y z	
	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	
Numeric	0 1 2 3 4 5 6 7 8 9	
Special	= Equals	, Comma
	+ Plus	. Decimal point
	- Minus	$ Currency symbol
	* Asterisk	' Apostrophe
	/ Slash	: Colon
	(Left parenthesis Space	
	) Right parenthesis	
HP Extensions	^ Circumflex	% Percent
	\ Backslash	? Question mark
	_ Underscore	" Quotation mark

Program Line Format

FORTRAN program lines are initial lines, continuation lines, comment lines, or compiler directives.

By default, each source line can be a maximum of 132 characters. FORTRAN treats characters beyond the maximum line length as comments. You can use the COLUMNS compiler directive to change the maximum length of a source line to any number of characters from 12 through 132.

The ANSI standard requires that source lines be exactly 72 characters. When the ANSI compiler directive is in effect, each line can be from 72 to 132 characters in length. If a line is shorter than 72 characters, the compiler appends blanks to make it 72 characters. If a line is longer than 72 characters, the compiler treats characters beyond position 72 as comments.

The ANSI directive differs from the COLUMNS 72 directive. When the ANSI directive is in effect, the compiler ensures that all source lines are at least 72 characters long by appending blanks to lines that contain fewer than 72 characters. If the COLUMNS 72 directive is in effect, the compiler does not append blanks to lines that have fewer than 72 characters. The difference between the COLUMNS 72 directive and the ANSI directive is important when a character constant is continued from one line to the next.

If your program specifies both an ANSI directive and a COLUMNS directive, the ANSI directive overrides the COLUMNS directive.

FORTRAN source lines contain the following elements:

Column	Meaning
1	If column 1 contains the letter “C” or an asterisk, the entire source line is a comment. If column 1 contains a question mark (?), the remainder of the record contains compiler directives. If column 1 contains a blank, the record is a continuation record or an optionally labeled FORTRAN statement. If column 1 contains any character other than a “C”, an asterisk, or a digit from 0 through 9, FORTRAN reports a syntax error.
1 through 5	If column 1 does not contain a “C” or an asterisk, columns 1 through 5 contain a label. The label consists of any combination of the digits 0 through 9. Blanks, if present, are ignored. If columns 1 through 5 are blank, the record does not contain a label.
6	If column 6 contains any character other than a zero (0) or a blank, column 7 is the beginning of a continuation line. If column 6 is a zero or a blank, column 7 is the beginning of a new FORTRAN statement.
7 through end of line	If column 1 specifies a comment, columns 7 through the end of the line are comment text. If column 6 is a zero or a blank, column 7 is the beginning of a new FORTRAN statement. If column 6 is any character other than a zero or a blank, column 7 is the beginning of a new FORTRAN statement.

Initial Line

A statement label, if present, begins anywhere in columns 1 through 5 (blanks are ignored). Column 6 can be a blank or a zero (0). FORTRAN statements begin in column 7.

Continuation Line

Continuation lines enable you to continue a statement beyond the limits of a physical line. You designate a continuation line by placing any character, except a 0 or a blank, in column 6. Begin the text of a continuation line in or after column 7. Following an initial line, you can use up to 19 continuation lines to write one FORTRAN statement.

Comment Line

Designate comment lines by specifying the character C or an asterisk (*) in column 1. The compiler treats all characters from column 2 to the end of the line as a comment. You can place comment lines anywhere in a program. A line that contains all blanks is a comment line. You can use blank lines to improve readability.

You cannot use continuation lines to continue comments. Write multi-line comments as multiple comment lines, as shown in the following example:

```
* This is such a lengthy comment line that I am afraid
* I shall have to continue it over several lines.
```

Compiler Directives

Designate compiler directives by placing a question mark (?) in column one, followed by the directive name. For example:

```
?SYNTAX
DIMENSION A(100),B(100)
COMMON A
.
100 STOP
END
```

You can write more than one directive on a line by inserting a comma between directives. The following directive lines are equivalent:

```
?ANSI, PAGE "This is the first page"
```

```
?ANSI
?PAGE "This is the first page"
```

Some directives can occupy multiple lines. Begin each line that continues a directive with a question mark in the first column. For example:

```
?SOURCE routines (probability, random, trig, volume,
?combinations, permutations)
```

Note that some directives must be written last on a directive line. For more information about using directives, see Section 10, Compiler Directives. The compiler ignores blanks in directives, except within character constants.

Treatment of Blanks in a Program Line

The FORTRAN compiler ignores blanks in columns 1 through 5, and in columns 7 through the end of the line of initial and continuation lines of all statements, except within character and Hollerith constants. The following two statements are equivalent; both assign the value 5 to the variable READN:

```
read n = 5
readn=5
```

Example 2-1. Sample FORTRAN Program: Program Lines

```
C   This program converts Fahrenheit to Celsius.
C
C   It reads an initial Fahrenheit value (i), a terminal
C   Fahrenheit value (j), and an increment value (k) from the
C   terminal. Then it prints a table showing the
C   corresponding Celsius values.
C
C   Set up table for titles, headings, and entries

?SYNTAX
?LIST, CODE
     PROGRAM conversion
     WRITE (6,50)
     WRITE (6,60)

50   FORMAT (x,'TABLE SHOWING TEMPERATURE CONVERSION FROM
    + FAHRENHEIT TO CELSIUS')

60   FORMAT (5X, 'Fahrenheit', 4X, 'Celsius')
70   FORMAT (5X, I4, 10X, F6.2)

C   Set up double loop: inner for computation of Celsius
C   temperature, outer to supply values for I, J, and K.
      DO 200 kount = 1,2
         READ *, i, j, k
         DO 100 fahrenheit = i, j, k
            celsius = 5./9. * (fahrenheit - 32)
            WRITE (6, 70) fahrenheit, celsius
100      CONTINUE
200   CONTINUE
      END
```

Symbolic Names

You use symbolic names to represent the following entities:

- Main program name
- Common block name
- Block data subprogram name
- Subroutine name
- External function name
- Variable name
- Array name
- RECORD and RECORD-field name
- Symbolic constant name
- Intrinsic function name
- Statement function name
- Dummy procedure name

Symbolic names can contain the letters A through Z, the numbers 0 through 9, and the underscore character (_). References to RECORD fields can also contain the circumflex character (^).

A symbolic name can be up to 31 characters long. The first character of a symbolic name must be a letter. The symbolic name can include blanks (in addition to the 31 characters), but the compiler ignores them. The compiler also ignores case. The compiler treats the following names identically:

```
MATHROUTINES        Math Routines
mathroutines        math routines
```

The following names are valid FORTRAN symbolic names:

```
a1b4300891
visitors march 1985 with free passes
```

The following names are invalid:

```
85 taxes      <-- name cannot begin with a digit
amount$       <-- name cannot contain a dollar sign
```

Context determines whether a particular sequence of characters identifies a keyword or a symbolic name. No sequence of characters is reserved in all contexts in FORTRAN.

Scope of Symbolic Names

The scope of a symbolic name is the range within which the symbolic name is defined: a name's scope can be an executable program, a program unit, a statement function statement, or an implied DO list in a DATA statement.

- The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have the scope of an executable program.
- The names of variables, arrays, RECORDs, RECORD fields, constants, statement functions, intrinsic functions, and dummy procedures have the scope of a program unit.
- The names of variables that appear as dummy arguments in a statement function statement have the scope of that statement.
- The names of variables that appear as the DO variable of an implied DO in a DATA statement have the scope of the implied DO list.

No two entities in the same scope can have the same symbolic name, except that a common block name can be the same as a program unit name.

If you use the name of a FORTRAN intrinsic function as a symbolic name, all references to that name within the program unit that declares the symbolic name reference your identifier name. You cannot invoke the intrinsic function from that program unit.

Avoid using FORTRAN keywords as symbolic names. Although the FORTRAN compiler does not report a warning or an error if you use a keyword as a symbolic name, your program is less readable.

Data Types

Every variable, array, RECORD field, symbolic constant, statement function, and function name has a type. FORTRAN recognizes the following data types:

- Integer
- Real
- Double precision
- Complex
- Logical
- Character

The symbolic name of a main program, subroutine, common block, or block data subprogram does not have a data type.

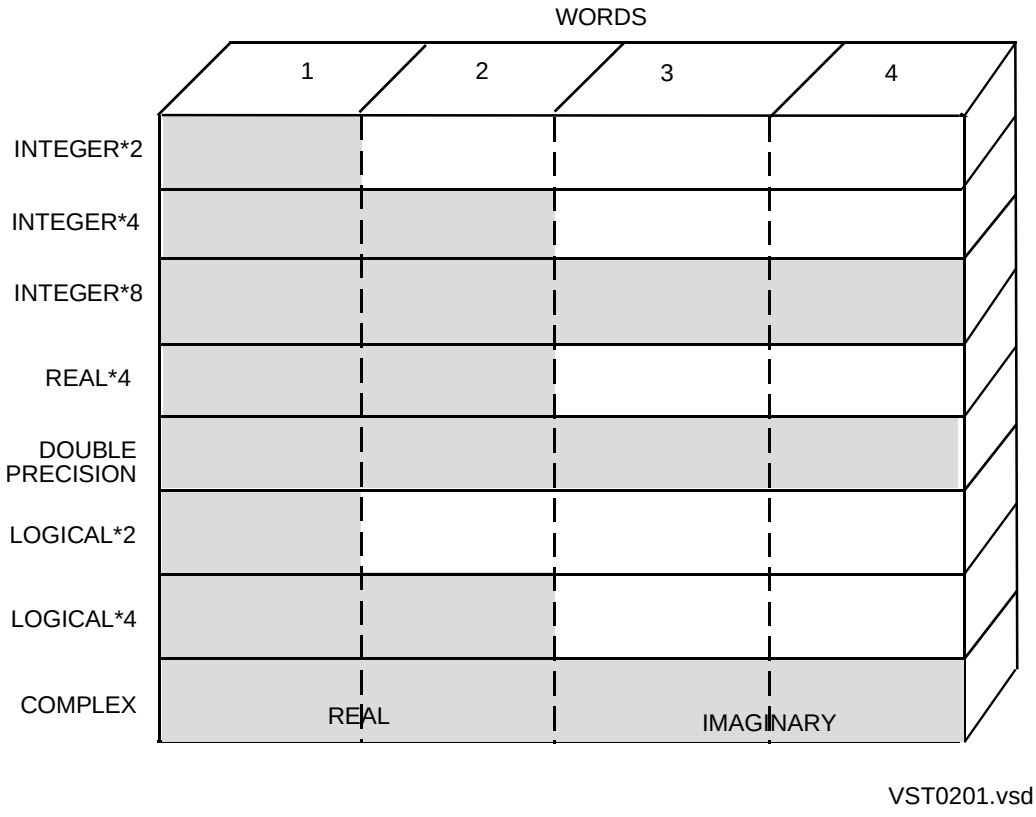
The data types INTEGER*4 and INTEGER*8, and the compiler directives INTEGER*4, INTEGER*8, and LOGICAL*4 enable you to allocate additional storage for large integers and logical type data.

Table 2-2. FORTRAN Data Types

Type	Range/Storage	Precision (digits)
Integer*2	-32,768 +32,767	
Integer*4	-2,147,483,648 +2,147,438,647	
Integer*8	-9,223,372,036,854,775,808 +9,223,372,036,854,775,807	
Real	±8.646000 E-78 ±1.579208 E+77	6.9
Double Precision\	±8.6361685550944446 E-78 ±1.15792089237316189 E+77	16.5
Complex	±8.6361685550944446 E-78 ±1.15792089237316189 E+77	6.9
Logical*2 (default)	word storage	
Logical*4 (directive)	doubleword storage	
Character	255 string length 1/2 word per character	

Figure 2-1 on page 2-9 shows the storage allocated for each type.

Figure 2-1. FORTRAN Data Storage



Implicit and Explicit Typing

By default, FORTRAN interprets symbolic names that begin with the letters I, J, K, L, M, and N as type integer and interprets names that begin with any other letter as type real. You can override the default type by using one of the following type declaration statements:

```
INTEGER*2
INTEGER*4
INTEGER*8
REAL
DOUBLE PRECISION
LOGICAL
COMPLEX
CHARACTER
```

The INTEGER type declaration, without an explicit length, refers to the prevailing integer type as determined by an INTEGER*2, INTEGER*4, or INTEGER*8 compiler directive. In the absence of such a directive, INTEGER means INTEGER*2. Similarly,

the LOGICAL type declaration refers to the prevailing logical type as determined by the LOGICAL*4 compiler directive. In the absence of such a directive, LOGICAL means LOGICAL*2.

The general form of a type declaration statement is:

```
type var-name [, var-name ]...
```

type

is a declaration statement name such as INTEGER*2 or COMPLEX

var-name

is the symbolic name of a constant, variable, array, RECORD field or function

For example, the following statement declares a double precision variable called INVENTORY:

```
DOUBLE PRECISION inventory
```

You can use an IMPLICIT statement to change the default type associated with the letters of the alphabet. For example, the following statement reverses the default type setting:

```
IMPLICIT REAL (i-n), INTEGER (a-h, o-z)
```

Data Storage—Standard Conformance

The ANSI standard requires that integer, real, and logical variables occupy the same amount of storage space. To write a program that conforms to the standard, begin the program with INTEGER*4 and LOGICAL*4 directives, and avoid using the INTEGER*n form to declare variables and other symbolic names. For example:

```
PROGRAM main
?INTEGER*4, LOGICAL*4
INTEGER patient number   <-- Declares INTEGER*4 variable
LOGICAL discharged       <-- Declares LOGICAL*4 variable
.
END
```

Constants

A constant is an unvarying datum. A constant can be a number, a constant expression, a complex value, a logical value, or a string of characters.

You can use a PARAMETER statement to create a symbolic name for a constant. See the PARAMETER Statement on page 7-79.

Arithmetic Constants

Integer, real, double precision, and complex constants are called arithmetic constants.

Integer Constants

You must express an integer constant as a whole number. You can place a minus sign in front of an integer to indicate a negative number. The following are examples of integer constants:

```
123           <-- word integer
-52381        <-- doubleword integer
9814387278    <-- quadrupleword integer
```

An integer constant is considered to be of the smallest size (INTEGER*2, INTEGER*4, INTEGER*8) that can contain it.

An integer constant cannot contain a decimal point or a comma. However, you can make a constant more readable by grouping digits of the constant and separating the groups with blanks. The compiler ignores the blanks. For example:

```
9 814 387 278
```

You can represent an integer constant in octal notation. An octal constant has the range of a quadrupleword integer and is a string of from 1 to 22 digits—only the digits 0, 1, 2, 3, 4, 5, 6, and 7 are valid—prefixed with a percent sign (%). Precede the percent sign with a minus sign if the number is negative. The following is an example of a negative, octal integer.

```
-%1034621
```

Real Constants

A real constant is a string of decimal digits that must include a decimal point or an exponent. It cannot include commas, but you can use embedded spaces for readability. You can express a real constant as a decimal number. For example:

```
7.5
-.0001
24578.342
```

You can also express a real constant in “exponential” form:

where *coefficient* is a decimal integer or real constant, and *exponent* is a decimal integer constant. For example:

Exponential Notation	Value
2.5E2	250
-1E-5	-.00001
-.00028E5	-28

The number following the letter E designates a power of 10. For example:

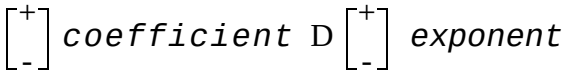
```
3.2E2    is the same as 3.2 * 10**2
1.23E-3  is the same as 1.23 * 10**-3
```

The following real constants are invalid:

```
2,345,125     <-- Number must not include commas
$3.21E5       <-- Number must not include special characters
23E83         <-- Number is too large
```

Double Precision Constants

A double precision constant is a quadrupleword real constant. Write a double precision constant like a real constant, but use the letter D to indicate the exponent part of the constant, as shown in the following syntax:



This shows examples of double precision constants:

Exponential Notation	Value
5.834D2	583.4
3122D5	312,200,000
14.D-6	.000014

Complex Constants

You express a complex constant as a pair of real or integer constants separated by a comma and enclosed in parentheses:

```
( real, imaginary)
```

where *real* is a real or integer constant that specifies the real part, and *imaginary* a real or integer constant that specifies the imaginary part. For example:

```
(1, 7.5)
(5, 1)
(-2.5E3, 3.67)
```

Logical Constants

There are two logical constants:

```
.TRUE.
.FALSE.
```

You must write the enclosing decimal points. A logical value is stored in one 16-bit word. The LOGICAL*4 compiler directive provides doubleword logical values. The only values used for logical constants are all bits set to 1 for .TRUE., and all bits set to for .FALSE..

Character Constants

A character constant is a string of up to 255 characters enclosed in apostrophes ('). A space character in a string occupies a position and is therefore significant. The following is an example of a character constant:

```
'April 1985'
```

If the string contains an apostrophe, you must use two apostrophes to distinguish it from the terminating apostrophe. The compiler treats two adjacent apostrophes as one apostrophe.

```
'Mozart''s "Don Giovanni"'
'The ''customer'' is always right!'
```

The following character constants are invalid:

```
"Report Summary" <-- The constant is not enclosed in
apostrophes.
'Sam's Diner' <-- The inner apostrophe is not doubled.
```

Character constants are case sensitive—the compiler retains uppercase and lowercase letters exactly as you specify them in the character constant.

You can use Hollerith constants to represent character data. You might need to use Hollerith data if you are working with pre-FORTRAN 77 programs. For additional details, see Appendix H, Hollerith Constants and Punch Card Codes.

Variables

A variable names a storage location whose contents can change during program execution. You identify a variable by a symbolic name.

You can use a declaration statement to declare explicitly the data type of a variable. For example:

```
INTEGER*8 employee
REAL balance, tax
DOUBLE PRECISION tonnage (12, 365)
```

If the variable is type character, you must specify its length when you declare its type. For example:

```
CHARACTER name*15, address*20, city*10, state*4, zip*9
```

The preceding CHARACTER declaration creates five variables whose lengths are 15, 20, 10, 4, and 9 bytes respectively. FORTRAN allocates one byte of storage for CHARACTER variables that do not include a length specification.

The range, precision, and storage allocation for variables is the same as for constants.

Arrays

An array is a sequence of elements of the same type, identified by one symbolic name.

Dimensioning an Array

A FORTRAN array has a minimum of one dimension and a maximum of seven dimensions. You can specify the dimensions of an array using a DIMENSION, COMMON, RECORD, or type declaration statement. When you declare an array, you must also declare its dimensions in the following form:

```
array-name ( d [ , d ]... )
```

array-name is the name of the array and d specifies the bounds of an array dimension and takes the form:

```
[ lower : ] upper
```

lower specifies the lower bound of the dimension. The lower bound can be zero, negative, or positive. If you do not specify a lower bound, the compiler uses 1 as the lower bound.

upper specifies the upper bound of the dimension. The upper bound can be zero, negative, or positive. The upper bound must be greater than or equal to the lower

bound. The number of elements in each dimension is one greater than the difference between the upper and lower bounds.

The following statement declares and dimensions the array DICTIONARY, a one dimensional character array. The first element is DICTIONARY(1) and the last element is DICTIONARY(100). Each of the 100 elements of DICTIONARY consists of 10 characters.

```
CHARACTER * 10 dictionary (100)
```

The following statement declares and dimensions the array CUSTOMERS. FORTRAN determines the array type implicitly as real. The array has seven dimensions—the maximum for a FORTRAN array:

```
DIMENSION customers (31, 5, 24, 52, 2, 1, 165)
```

Table 2-3 shows the organization of the array declared by the statement:

```
DIMENSION ta (6,6)
```

Table 2-4 shows the organization of the array declared by the statement:

```
DIMENSION tb (-3:2,-2:3)
```

Note that TA and TB are exactly the same size. TA, however, is based at 1,1 whereas TB is based at -3,2.

Table 2-3. Array Based at One

ta(1,1)	ta(1,2)	ta(1,3)	ta(1,4)	ta(1,5)	ta(1,6)
ta(2,1)	ta(2,2)	ta(2,3)	ta(2,4)	ta(2,5)	ta(2,6)
ta(3,1)	ta(3,2)	ta(3,3)	ta(3,4)	ta(3,5)	ta(3,6)
ta(4,1)	ta(4,2)	ta(4,3)	ta(4,4)	ta(4,5)	ta(4,6)
ta(5,1)	ta(5,2)	ta(5,3)	ta(5,4)	ta(5,5)	ta(5,6)
ta(6,1)	ta(6,2)	ta(6,3)	ta(6,4)	ta(6,5)	ta(6,6)

Table 2-4. Array With Negative Subscripts

tb(-3,-2)	tb(-3,-1)	tb(-3,0)	tb(-3,1)	tb(-3,2)	tb(-3,3)
tb(-2,-2)	tb(-2,-1)	tb(-2,0)	tb(-2,1)	tb(-2,2)	tb(-2,3)
tb(-1,-2)	tb(-1,-1)	tb(-1,0)	tb(-1,1)	tb(-1,2)	tb(-1,3)
tb(0,-2)	tb(0,-1)	tb(0,0)	tb(0,1)	tb(0,2)	tb(0,3)
tb(1,-2)	tb(1,-1)	tb(1,0)\	tb(1,1)	tb(1,2)	tb(1,3)
tb(2,-2)	tb(2,-1)	tb(2,0)	tb(2,1)	tb(2,2)	tb(2,3)

Array References

You can reference an entire array using the array name, or you can reference a specific array element using the array name followed by a subscript. A reference to an array element takes the following form:

```
name ( e [, e ]... )
```

where *name* is the symbolic name of the array and e is an integer subscript expression. For example:

```
CUSTOMERS (2,3,100)
```

You must specify a value for each dimension of an array when you reference an element of that array. The following reference is invalid because it specifies only two dimensions of a three-dimensional array:

```
REAL inventory(10,31,365)
inventory (1,1) = 234.52
```

If you refer to an entire array, FORTRAN accesses the elements in the order indicated in Figure 2-2 on page 2-18.

A subscript can contain function references, but the evaluation of the function must not alter the value of any other subscript expression in the array. In the following example, the subscript used in the reference to EVEN NUMBERS includes the variable J:

```
DIMENSION even numbers (100)
DO 20 j = 1,100
even numbers (j) = j * 2
20 CONTINUE
```

You can calculate the storage location of an array element using the formulas in Table 2-5.

Table 2-5. Calculating Array Element Storage Locations (page 1 of 2)

Dimensions	Position of Array Element
1	$1 + (s_1 - j_1)$
2	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1$
3	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1 + (s_3 - j_3) * n_1 * n_2$
4	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1 + (s_3 - j_3) * n_1 * n_2 + (s_4 - j_4) * n_1 * n_2 * n_3$
.	.
.	.
.	.
7	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1 + (s_3 - j_3) * n_1 * n_2 + (s_4 - j_4) * n_1 * n_2 * n_3 + (s_5 - j_5) * n_1 * n_2 * n_3 * n_4 + (s_6 - j_6) * n_1 * n_2 * n_3 * n_4 * n_5 + (s_7 - j_7) * n_1 * n_2 * n_3 * n_4 * n_5 * n_6$

Table 2-5. Calculating Array Element Storage Locations (page 2 of 2)

Dimensions	Position of Array Element
j_i	Lower bound of dimension i.
k_i	Upper bound of dimension i.
n_i	Size of dimension i. If the lower bound is one, $n_i = k_i$. Otherwise, $n_i = (k_i - j_i + 1)$.
s_i	Value of the subscript expression specified for dimension i.

Array Size

The number of elements in an array is equal to the product of the number of elements in each dimension of the array.

Storage Order

FORTRAN stores array elements as a linear sequence of words. The type and the number of elements in an array determine the number of words of memory FORTRAN reserves for the array.

FORTRAN reserves one word of storage for each element in an array of one-word elements, two times the number of elements for an array of doubleword elements, and four times the number of elements for an array of quadrupleword elements. For character arrays, the number of storage words allocated equals half the number of characters in each element of the array times the number of elements in the array.

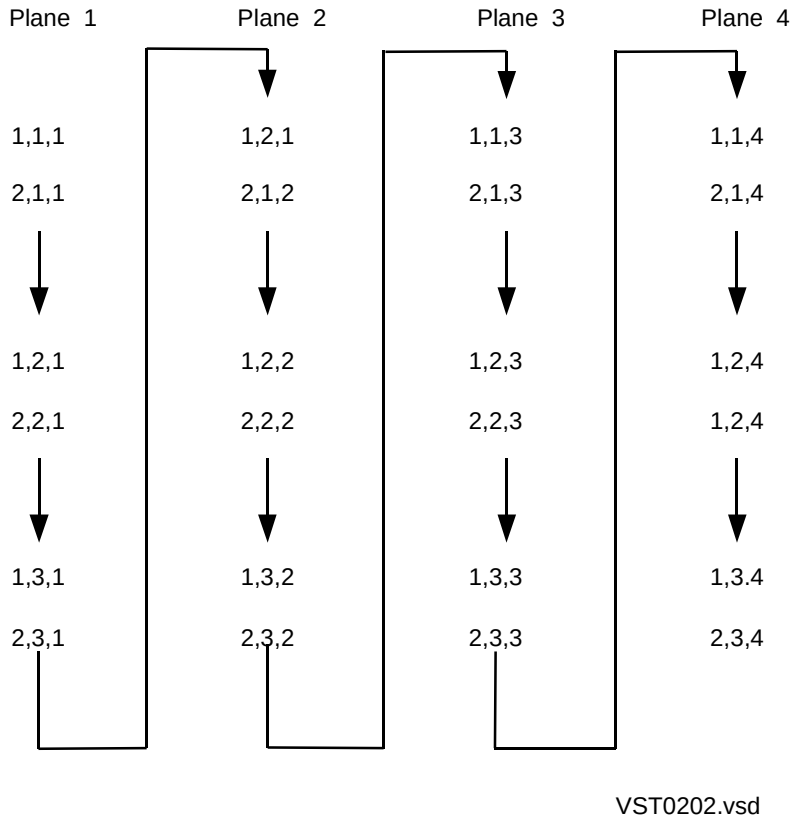
The following array requires storage for 200 characters or 100 storage words:

```
CHARACTER * 10 product (20)
```

FORTRAN stores array elements in ascending locations by columns. The first (leftmost) subscript increases most rapidly, the last (rightmost) subscript increases least rapidly. Figure 2-2 on page 2-18 shows the storage order for the following array:

```
DIMENSION income(2,3,4)
```

Figure 2-2. Storage Order of Array Elements



To find the position of EMPLOYEE(-2, 7, 3) in the array EMPLOYEE(-5:5, 8, 5):

```
pos =  1 + (-2 -(-5))
         + (7 - 1) * 11
         + (3 - 1) * 11 * 8
pos =  1 + 3 + 66 + 176
pos =  246
```

The following example shows how to find the position of a character in a character array. For example, to find ADDRESS(1, 52), you first determine the array element in which the character appears:

```
CHARACTER*10 address(2,100)
array-element = 1 + (1 - 1)
                    + (52 - 1) * 2
array-element = 103
```

Second, using the array-element you just calculated, determine the byte offset of the character relative to the beginning of the array. The character storage position is equal to:

```
1 + (array-element -1) * character-length
1 + 102 * 10
1021
```

Considerations

- Use a DIMENSION, type-declaration, or COMMON statement to declare the number of dimensions and the number of elements in each dimension of an array.
- You must declare the array before the first reference to the array.
- Declare the array type explicitly unless you want the array to be typed by default.
- Do not use subscript values that are outside the range of the declared minimum and maximum bounds of the array. For additional information, see the BOUNDSCHECK Compiler Directive on page 10-8.
- You can use an arithmetic expression that evaluates to an integer to determine a subscript value.
- If you use an unsubscripted array name, the compiler assumes that you are referencing the entire array.
- You cannot redefine the dimensions or size of an array within an executable program.

For a discussion of assumed-size and adjustable-size array declarators, see Section 4, Program Units.

Substrings

You can reference a character-type variable in its entirety or in part. To reference a part of a character-type variable, you must append a subscript reference to the variable name. A subscript reference uses the form:

```
var-name ( [ first ] : [ last ] )
```

var-name

is the name of the variable

first

specifies the position of the first character in the substring

last

specifies the position of the last character in the substring

For example, if you store the string ABRACADABRA in the variable PASSWORD, the substring

```
password (3:6)
```

refers to the characters RACA.

If you omit first, the substring begins with the first character in var-name. If you omit last, the substring ends with the last character in var-name. For example:

```
password (:5)
```

refers to the characters ABRAC;

```
password (7:)
```

refers to the characters DABRA.

You can reference a substring of an array element in a character array by specifying the array element, followed by the substring reference. For example, if the fifth element of the array GAMES(100) contains the string HANGMAN, the following substring reference contains the characters HANG.

```
games(5)(1:4)
```

Records

The RECORD declaration statement is a HP extension that enables you to declare structured data. Unlike arrays, the elements of RECORDs can be different types. The following is an example of a RECORD declaration:

```
RECORD addresses           <-- addresses is the RECORD name
  FILLER*5
  CHARACTER*10 lastname    <-- data type declaration
  CHARACTER*10 firstname       (RECORD field name)
  CHARACTER*1 middle
  FILLER*2                 <-- FILLER declaration
  CHARACTER*20 street
  CHARACTER*10 city
  CHARACTER*2 state
  CHARACTER*9 zip
END RECORD
```

Records defined in RECORD declaration statements are called RECORDs. The individual elements of a record are called RECORD fields.

You can use the Data Definition Language (DDL) to share FORTRAN RECORD definitions between FORTRAN modules. For more information, see the *Data Definition Language (DDL) Reference Manual*.

Writing a RECORD Declaration

A RECORD declaration must begin with a RECORD statement and end with an END RECORD statement. You can declare a dimensioned RECORD, but a dimensioned RECORD cannot have more than one dimension.

The fields of a RECORD declaration can be:

- Data type declarations
- FILLER declarations
- RECORD declarations
- EQUIVALENCE statements

A data type declaration, including an array declaration, is an elementary field of a RECORD. An elementary field is a field that is not itself a RECORD. RECORD fields that are arrays can have only one dimension:

```
RECORD a
  INTEGER b(20)        <-- One-dimensional array is OK
  INTEGER c(10, 20)    <-- Two-dimensional array is NOT valid
END RECORD
```

A FILLER declaration defines a specified number of one-byte pad characters within a RECORD and enables you to align character positions when you equivalence RECORDs. You write a FILLER declaration in the following form:

```
FILLER * number
```

where *number* specifies the number of pad characters. number cannot exceed 255.

A RECORD declaration can include nested records. A nested RECORD is a nonelementary field of a RECORD and has the properties associated with data of type character. A nested RECORD can have at most one dimension. You can nest RECORDs within RECORDs to a maximum depth of 15 levels.

An EQUIVALENCE statement in a RECORD declaration can refer to any data item that appears at the same nesting level as the EQUIVALENCE statement. You must refer to any equivalenced entity as a unit. You can equivalence array names or character variable names but not array elements or substrings.

Referencing a RECORD Field

In an executable statement, you refer to a field within a RECORD by appending a circumflex to the end of the RECORD name followed by the name of the field that you want to reference. You might have a succession of circumflex/field-name combinations if you reference a field of a RECORD that is nested more than one level deep.

```
record-name^ field-name [ ^ field-name ]...
```

record-name

is the name of the outermost RECORD.

field-name

is the name of a field within a RECORD. field-*name* can be any field within a RECORD except a FILLER.

In the following example, the syntax to reference each field of the employee RECORD appears to the right of the field:

```
RECORD employee              <-- employee
  FILLER*10                  <-- cannot be referenced
  CHARACTER*10 name          <-- employee^name
  CHARACTER*10 hired         <-- employee^hired
  RECORD salary              <-- employee^salary
    REAL pay                 <-- employee^salary^pay
    INTEGER dept             <-- employee^salary^dept
  END RECORD
END RECORD
```

You must always fully qualify a field name to access it.

You cannot access a FILLER field.

The following example includes references to fields in the employee RECORD:

```
CHARACTER*10  emp_name
REAL          emp_pay
INTEGER       emp_dept

emp_name = employee^name
emp_pay =  employee^pay     <-- Error: Not fully qualified
emp_pay =  employee^salary^pay
emp_dept = employee^salary^dept
emp_temp = employee^filler  <-- Error: Cannot refer to FILLER
```

RECORD Storage

FORTRAN allocates the fields of a RECORD in the order in which they appear in the RECORD declaration. It allocates to each field the amount of storage required by the field's type. If the field is an array, the amount of storage is the size of an element of the array multiplied by the number of elements in the array. A field of type character or RECORD is allocated beginning at the next free byte. A field of type logical or numeric is aligned at the next word boundary. If a RECORD itself is dimensioned and contains at least one field of type logical or numeric, FORTRAN ensures that the record starts at an even-byte address. A RECORD at the outermost level always begins on a word boundary.

Equivalencing RECORDs

You can equivalence RECORDs only to other RECORDs. You can use an EQUIVALENCE statement:

- Within a RECORD declaration to equivalence any data items at the same nesting level as the EQUIVALENCE statement.
- Outside a RECORD declaration, to equivalence RECORDs only at their outermost level.

The following example shows a nested RECORD declaration that contains two equivalenced subrecords:

```
RECORD dependent
  RECORD depdata1
  FILLER*5
    RECORD depfields
      CHARACTER*10 firstname
      CHARACTER*20 lastname
      CHARACTER*2 birthday
      CHARACTER*2 birthmonth
      CHARACTER*2 birthyear
    END RECORD
  END RECORD
  RECORD depdata2
    RECORD depkey
      CHARACTER*5 employeenum
    END RECORD
    FILLER*36
  END RECORD
  EQUIVALENCE (depdata1, depdata2)
END RECORD
```

Equivalencing RECORD Fields

You can declare two or more fields in a RECORD that share the same storage by referencing the items in an EQUIVALENCE declaration. The items need not be the same data type. You can equivalence character and numeric variables only within a

RECORD. You cannot equivalence character and numeric variables in declarations outside of a RECORD.

```
?GUARDIAN OPEN
      RECORD file
        INTEGER*2 name(12)
        CHARACTER namestring * 24
        EQUIVALENCE (name, namestring)
      END RECORD
      file^namestring = '$data subvol myfile '
      CALL open (file^name, n, %2001)
```

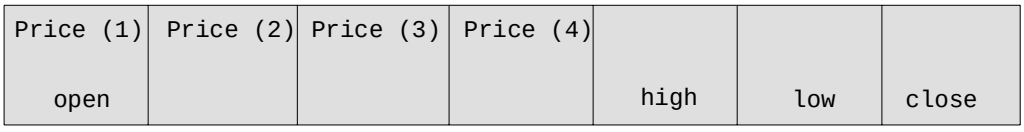
If two fields of different sizes are equivalenced within a RECORD, the next field begins after the larger of the two equivalenced fields. For example, in the RECORD declaration:

```
RECORD stock
  REAL open, high, low, close
  REAL price (4)
  EQUIVALENCE (price, open)
END RECORD
```

the field STOCK^HIGH is not in the same location as the second element of the STOCK^PRICE array, but instead follows the fourth element of the STOCK^PRICE array. Figure 2-3 illustrates how the fields of the STOCK RECORD are allocated in memory.

Figure 2-3. Memory Allocation for Equivalenced Fields in a RECORD

```
RECORD stock
     REAL open high, low ,cose
     REAL price   (4)
     EQUIVALENCE (price, open)
END  RECORD
```



VST0203.vsd

Considerations

- You cannot use a DATA statement to initialize the value of a RECORD, or a field of a RECORD.
- You cannot declare a RECORD within a BLOCK DATA subprogram.
- A common block can contain at most one RECORD.
- A common block that contains a RECORD cannot contain variables or arrays.
- The maximum size of a RECORD, or of one element of a dimensioned RECORD, is 32,767 bytes, including all of its constituent fields.

3 Expressions

This section describes the syntax and semantics of FORTRAN expressions. An expression consists of operands, operators, and parentheses. The evaluation of an expression yields a single value, whose type is determined by the type of the operands.

Topics covered in this section include:

A FORTRAN expression is either simple or compound. A simple expression consists of a single element: a constant, variable, array element, or RECORD field. A compound expression consists of one or more operands and one or more operators.

FORTRAN recognizes four types of expressions.

- Arithmetic
- Character
- Relational
- Logical

FORTRAN also processes constant expressions. A constant expression contains only constants and symbolic constants. A constant expression can be arithmetic, character, or logical:

```
3 + 4                       <-- An arithmetic-type expression
'Rip' // 'Van' // 'Winkle'  <-- A character-type expression
.TRUE.                      <-- A logical-type expression
```

Arithmetic Expressions

The combination of arithmetic operands and operators makes up an arithmetic expression. An arithmetic expression expresses a numeric computation; the evaluation of an arithmetic expression produces a numeric value.

The syntax of an arithmetic expression is:

```
[ prefix-op ] item [ infix-op item ]...
```

prefix-op

is one of

+

-

item

is a variable, constant, symbolic constant, record field, function reference, or an arithmetic expression enclosed in parentheses.

infix-op

is one of:

**

*

/

+

-

If a signed element (*prefix-op item*) follows an infix operator, you must enclose the signed element in parentheses as in the following example:

```
A + (-B)
```

Table 3-1. Arithmetic Operators

Operator	Operation	Example	Meaning
**	Exponentiation	A ** B	Raise A to the power B
*	Multiplication	A * B	Multiply A and B
/	Division	A / B	Divide A by B
+	Addition	A + B	Add A and B
+	Identity	+ A	Positive A
-	Subtraction	A - B	Subtract B from A
-	Negation	- A	Negative A

Evaluation of Arithmetic Expressions

The hierarchy of arithmetic operators determines the order in which the operands are combined:

```
**          <-- Highest
* and /
+ and -     <-- Lowest
```

For example, in the following expression, the exponentiation operator takes precedence over the negation operator:

```
-A ** 5     <-- Evaluated as -(A ** 5)
```

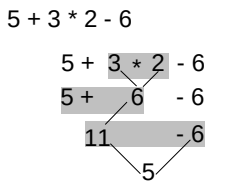
You can use parentheses to override the normal sequence of evaluation. Within a compound expression, FORTRAN evaluates an expression enclosed in parentheses first. If you use nested parentheses, FORTRAN evaluates the deepest level expression first. Within parentheses, evaluation proceeds in the normal sequence. Parenthetical expressions must be balanced: you must match every left parenthesis with a corresponding right parenthesis. FORTRAN uses the following rules to evaluate expressions whose value depends upon precedence of operators:

- If adjacent operators have different precedence, the operator with higher precedence is evaluated first.
- Adjacent operators that have the same precedence are evaluated according to their associativity: right-to-left for the exponentiation operator, left-to-right for all others.

The preceding rules specify the order in which operators are executed, but not the order in which an operator's operands are evaluated. Thus, for example, the expression A + B + C is evaluated as (A + B) + C, but the compiler could evaluate C before it evaluates (A + B). If evaluating an operand can have side effects, for example by calling a function, the order in which the operands are evaluated can be important.

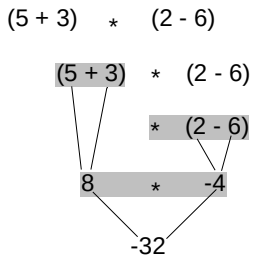
The following examples illustrate how HP FORTRAN evaluates expressions. The shaded portion at each step, consisting of an operator and two operands, shows the operator evaluated at that step.

- Example 1



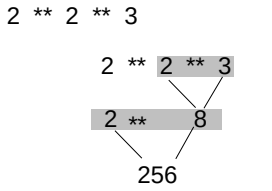
VST0301.vsd

- Example 2



VST0302.vsd

- Example 3



VST0303.vsd

Determination of Result Type

FORTRAN determines the data type of an arithmetic expression according to the data types of its operands.

If all the operands of an expression are the same type, the resulting value is also of that type.

If operands of different types appear in an expression, the data type of the result is determined by conversion rules that FORTRAN applies to the intermediate results generated during the evaluation of the expression. Table 3-2 shows how FORTRAN determines the type of an arithmetic expression involving two operands.

Table 3-2. Determination of Expression Type (page 1 of 2)

	X2			
x1	**I2**	**R2**	**D2**	**C2**
I1	I = I1 ~ I2	R = REAL(I1) ~ R2	D = DBLE(I1) ~ D2	C = CMPLX(REAL(I1),0. ~ C2
R1	R = R1 ~ (REAL(12)	R = R1 ~ R2	D = DBLE(R1) ~ D2	C = CMPLX(R1,0.) ~ C2

Table 3-2. Determination of Expression Type (page 2 of 2)

	X2			
x1	**I2**	**R2**	**D2**	**C2**
D1	D = D1 ~ DBLE(I2)	D = D1 ~ DBLE(R2)	D = D1 ~ D2	Illegal
C1	C = C1 ~ CMPLX (REAL(I2),0.)	C = C1 ~ CMPLX(R2,0.)	Illegal	C = C1 ~ C2

REAL, DBLE, CMPLX are type-conversion functions as described in Section 8, Intrinsic Functions
~ is +, -, *, or /.
I = INTEGER, R = REAL, D = DOUBLE PRECISION, C = COMPLEX.

The result of integer division is the signed, nonfractional part of the quotient. The fractional part is truncated, not rounded. For example, the values of K after the following assignments are 0:

```
K = 2/3 + 1/3     <-- = 0 + 0
K = -2/3          <-- = 0
```

Exponentiation entails additional restrictions. Table 3-3 shows the results when operands of different types are combined in an exponential expression.

Table 3-3. Evaluation of Mixed-Type Exponential Expressions

	Exponent			
Base	**Integer**	**Real**	**Double Precision**	**Complex**
Integer	Integer	Real	Double Precision	Complex
Real	Real	Real	Double Precision	Complex
Double	Precision Double	Precision Double	Precision Double	Precision Illegal
Complex	Complex	Complex	Illegal	Complex
Negative	type of base	Illegal	Illegal	Complex

Character Expressions

A character expression is composed of character elements and the concatenation operator. The evaluation of a character expression produces a result of type character.

The syntax of a character expression is:

```
char-element [ // char-element ]...
```

char-element

is a constant, symbolic constant, variable, RECORD name, array element, substring reference, function reference, or another character expression. char-element must be of type character.

//

is the concatenation operator.

The result of a character expression is a string, whose length is equal to the combined lengths of all the character elements appearing in the expression. The following constant character expression:

```
'Tomorrow'//' and tomorrow'//' and tomorrow--'
```

has the value

```
Tomorrow and tomorrow and tomorrow--
```

A character expression can contain up to 64 operands. A character expression cannot contain an operand whose length specification is an asterisk in parentheses, unless the operand is a symbolic constant.

Parentheses do not affect the value of a character expression. The following expressions are identical:

```
name // city // zip
name // (city // zip)
```

Relational Expressions

A relational expression compares the values of two arithmetic or two character operands. Relational expressions can appear only within logical expressions. The evaluation of a relational expression produces a logical type result, whose value is either true or false.

The syntax of a relational expression is:

```
expression rel-operator expression
```

expression

is an arithmetic or character expression.

rel-operator

is one of

.LT.

.LE.

.EQ.

.NE.

.GT.

.GE.

A relational expression has a value of .TRUE. only if the values of the operands satisfy the relation specified by the operator. Its value is .FALSE. otherwise.

Table 3-4. Relational Operators

Operator	Operation	Example	Meaning
.LT.	Less than	A .LT. B	Is A less than B?
.LE.	Less than or equal to	A .LE. B	Is A less than or equal to B?
.EQ.	Equal to	A .EQ. B	Is A equal to B?
.NE.	Not equal to	A .NE. B	Is A not equal to B?
.GT.	Greater than	A .GT. B	Is A greater than B?
.GE.	Greater than or equal to	A .GE. B	Is A greater than or equal to B?

Note: The enclosing decimal points are required.

In the following example, if the integer variables J and K have a value of 1 and 100 respectively, the relational expressions have the indicated values:

```
j .GT. k       <-- 1 .GT. 100 is false
k .GT. j       <-- 100 .GT. 1 is true
k .GE. j       <-- 100. GE. 1 is true
```

Evaluation of Relational Expressions

If a relational arithmetic expression contains operands of different types, FORTRAN converts the lower ranking data type to the higher ranking data type before comparing the operands.

A character string X is less than a character string Y if, starting at the left end of both strings, the first character in X that is not equal to the corresponding character in Y is less than the character in Y in the ASCII collating sequence. Similarly, a character string X is greater than a character string Y if, starting at the left end of both strings, the first character in X that is not equal to the corresponding character in Y is greater than the character in Y in the ASCII collating sequence.

If the character expressions being compared are different lengths, FORTRAN pads the shorter expression with trailing blanks until it is equal in length to the longer expression.

Considerations

- You cannot compare a COMPLEX value and a DOUBLE PRECISION value.
- You can use a COMPLEX value in an arithmetic relational expression only if the relational operator is .EQ. or .NE..

The following examples include valid relational expressions:

```
LOGICAL a, b
COMPLEX x, y
READ (*,*) w, x, y, z
a = x .EQ. y
b = w .GE. z

CHARACTER * 10 a, b
LOGICAL order
READ (*,*) a, b
order = a .GT. b
```

The following examples are invalid:

```
DOUBLE PRECISION y
COMPLEX x, z
LOGICAL a, b
READ (*,*) x, y, z
a = x .LE. y              <-- Cannot compare complex and
double precision values
b = z .GT. x              <-- Relational operator is not .EQ.
or .NE. and one of the operands is a complex value.
```

Logical Expressions

A logical expression designates a logical computation whose result is either true or false. The syntax of a logical expression is:

```
[.NOT.] logic-exp [ logic-op [.NOT.] logic-exp ]...
```

logic-exp

is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical symbolic constant, a logical variable, a logical array element reference, or a logical function reference.

logic-op

is one of:

```
AND.
.OR.
.EQV.
.NEQV.
```

Table 3-5 describes the meaning and use of the logical operators.

Table 3-5. Logical Operators

Operator	Operation	Example	Meaning
.NOT.	Negation	.NOT. x	Complement X
.AND.	Conjunction	x .AND. y	Boolean product of X and Y
.OR.	Inclusive disjunction	x .OR. y	Boolean sum of X and Y
.EQV.	Equivalence	x .EQV. y	X and Y are both true or both false
.NEQV.	Nonequivalence	x .NEQV. y	X is true and Y is false, or X is false and Y is true

If a logical expression contains two or more logical operators, FORTRAN uses the following hierarchy to determine the order in which the operators are evaluated:

```
.NOT.              <-- Highest
.AND.
.OR.
.EQV. or .NEQV.    <-- Lowest
```

The following expressions are equivalent:

```
x .OR. y .AND. z
x .OR. (y .AND. z)
```

You can use parentheses to override the normal order of precedence, as in the following example:

```
(x .OR. y) .AND. z
```

If an expression contains two or more adjacent .AND. operators, .OR. operators, .EQV. operators, or .NEQV. operators, FORTRAN evaluates the expression from left to right. Table 3-6 shows the results of combining two logical elements with the logical operators.

Table 3-6. Evaluation of Logical Expressions (page 1 of 2)

Operator	Operand	Operand	Result
.NOT.	.TRUE.		.FALSE.
	..FALSE.		..TRUE.
.AND.	.TRUE.	.TRUE.	.TRUE.
	.TRUE.	.FALSE.	.FALSE.
	.FALSE.	.TRUE.	.FALSE.
	.FALSE.	.FALSE.	.FALSE.
.OR.	.TRUE.	.TRUE.	.TRUE.
	.TRUE.	.FALSE.	.TRUE.
	.FALSE.	.TRUE.	.TRUE.
	.FALSE.	.FALSE.	.FALSE.
.EQV.	.TRUE.	.TRUE.	.TRUE.
	.TRUE.	.FALSE.	.FALSE.
	.FALSE.	.TRUE.	.FALSE.
	.FALSE.	.FALSE.	.TRUE.

Table 3-6. Evaluation of Logical Expressions (page 2 of 2)

Operator	Operand	Operand	Result
.NEQV.	.TRUE.	.TRUE.	.FALSE.
	.TRUE.	.FALSE.	.TRUE.
	.FALSE.	.TRUE.	.TRUE.
	.FALSE.	.FALSE.	.FALSE.

Operator Precedence

Each FORTRAN operator has a precedence relative to all other operators. When the compiler evaluates an expression, it compares the precedence of two adjacent operators. If one operator has higher precedence than the other, the operator with the higher precedence is evaluated first. For further information on operator precedence within an operator class, see the appropriate discussion in this section.

You can override the normal operator precedence by enclosing expressions in parentheses.

Table 3-7 shows the precedence of all FORTRAN operators, where 0 is the highest precedence, 9 is the lowest precedence.

Table 3-7. Operator Precedence (page 1 of 2)

Operator Class	Operator	Precedence
Arithmetic	**	0
	unary +	1
	unary -	1
	*	2
	/	2
	+	3
	-	3

Table 3-7. Operator Precedence (page 2 of 2)

Operator Class	Operator	Precedence
Character	//	4
Relational	.LT.	5
	.LE.	5
	.EQ.	5
	.NE.	5
	.GE.	5
	.GT.	5
Logical	.NOT.	6
	.AND.	7
	.OR.	8
	.EQV.	9
	.NEQV.	9

4 Program Units

This section describes the format of FORTRAN external procedures and BLOCK DATA subprograms. Topics covered in this section include:

The Main Program and Subprograms

A program unit consists of a sequence of statements and optional comment lines, and ends with an END statement. Each program unit is either a main program or a subprogram.

An executable FORTRAN program must contain exactly one main program unit. A main program is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It can have a PROGRAM statement as its first statement and must end with an END statement.

In mixed-language programs, the executable program's main routine might be written in a language other than FORTRAN. For more information, see Section 13, Mixed-Language Programming.

Your program begins executing with the first executable statement in your main program.

Subprograms other than BLOCK DATA are known collectively as procedures and include:

- External functions
- Subroutines
- Intrinsic functions
- Statement functions

The term external function refers to

- External functions specified in function subprograms
- External functions written in languages other than FORTRAN subprograms

The term subroutine refers to

- Subroutines specified in subroutine subprograms
- Subroutines written in languages other than FORTRAN

Intrinsic functions are described in Section 8, Intrinsic Functions. Statement functions are described in Section 7, Statements.

Section 13, Mixed-Language Programming, describes non-FORTRAN procedures that are either system procedures or procedures written in languages other than FORTRAN.

A FORTRAN procedure (or other subprogram) is a program unit that begins with a FUNCTION or SUBROUTINE statement, and ends with an END statement. You can compile subprograms separately from the main program. Section 9, Program Compilation, describes separate compilation.

FORTRAN also includes a nonexecutable subprogram called a BLOCK DATA subprogram that you can use to assign initial data values to COMMON entities.

Table 4-1. FORTRAN Program Units

Main Program	Subroutine Program	Function Subprogram	Block Data Subprogram
Executable	Executable	Executable	Nonexecutable
Not typed	Not typed	Typed implicitly or explicitly	Not typed
RETURN not allowed	Alternate RETURN allowed	RETURN allowed	RETURN not allowed
N.A.	Accepts values through arguments or common blocks	Accepts values through arguments or common blocks	N.A.
N.A.	Returns values through arguments or common blocks	Returns a value for the function name, or through arguments or common blocks	N.A.

Communication Between Program Units

A calling program unit and a referenced procedure can exchange data in common blocks and in arguments to the procedure. You can specify up to 63 arguments in a statement function or external function.

The calling program unit passes actual arguments to the referenced procedure. The arguments of the referenced procedure are called dummy arguments. When the referenced procedure executes, the actual arguments passed to it from the calling program unit replace the dummy arguments. To avoid error conditions, you must make sure that the type, order, and number of the actual and dummy arguments correspond.

- If the dummy argument is a variable name, it can be associated with an actual argument that is a variable, array element, substring, or expression. Note, however, that if information is passed back from the subprogram using a dummy argument, its associated actual argument cannot be an expression.

  If the associated actual argument is a constant or a symbolic constant, a function reference, an expression involving operators, or an expression enclosed in parentheses, the variable must not be redefined within the subprogram.

- If the dummy argument is an array name, it can be associated with an actual argument that is an array, array element, or array element substring. The size and dimensions of associated arrays can be different. If the actual argument is a noncharacter array name, the size of the dummy array must not exceed the size of the actual array. Each actual array element becomes associated with the dummy array element that has the same subscript value as the actual array element.

  If the actual argument is a noncharacter array element, the size of the dummy array must not exceed the size of the actual array plus one, minus the subscript value of the array element.

  Association by array elements exists for character arrays if the lengths of dummy and actual array elements are the same. If they are not, association still exists, but the dummy and actual array elements will not consist of the same characters.

- If the dummy argument is a RECORD name, it can be associated with a character array, character variable, or RECORD name.

- If the dummy argument is a procedure name and the dummy procedure is to be referenced as a function, the actual procedure must be the name of an intrinsic function, external function, or dummy procedure. If the name of the dummy function is the same as the name of an intrinsic function, that intrinsic function cannot be referenced in the same subprogram.

  If the dummy procedure is to be referenced as a subroutine, the actual procedure must be the name of a subroutine or dummy procedure.

  To use a procedure name as an actual argument, the name must appear in an EXTERNAL statement (for external procedures or dummy procedures) or in an INTRINSIC statement (for intrinsic functions) in the referencing program unit. For

additional information about the EXTERNAL and INTRINSIC statements, see Section 7, Statements.

Because you can compile FORTRAN program units independently, you must declare the data type of arguments both in the calling and referenced program unit, unless the arguments are typed by default.

Example

If you call the following function subprogram:

```
FUNCTION tinterest(principal, payment, rate, months)
.
END
```

using the statement,

```
total = tinterest(loans(cash, credit), pay, .06, 12 - x)
```

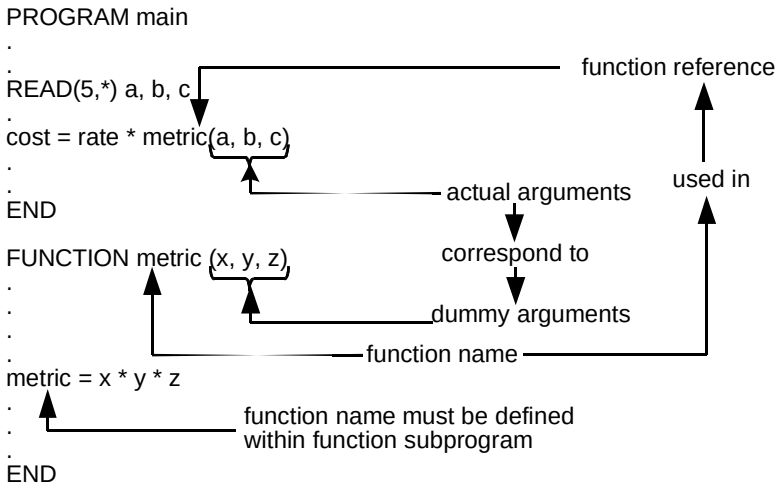
the execution of the function LOANS with arguments CASH and CREDIT provides the value of the first argument. The second argument is defined by the value of PAY. The constant .06 provides the value of the third argument, and the expression 12-X determines the value of the fourth argument.

Function Subprograms

A function subprogram returns a single value to the calling program unit. The calling program can be the main program, another function subprogram, a subroutine, or the function subprogram itself. As an extension to the ANSI standard, HP FORTRAN allows recursive functions.

A function subprogram executes when a calling program references its name in an expression. The function reference consists of the function name followed by an actual argument list, enclosed in parentheses. The parentheses are required also when there are no actual arguments. Figure 4-1 on page 4-5 shows the relationship between the function subprogram and the calling program.

Figure 4-1. Function Subprogram and Calling Program



When the function reference executes, FORTRAN evaluates any actual arguments that are expressions, and associates the actual arguments with the corresponding dummy arguments in the FUNCTION statement. Control then passes to the subprogram. The function subprogram must contain the following statements:

- A FUNCTION statement as its first statement
- A statement establishing a value for the function name
- An END statement as its last statement

The function subprogram cannot contain a PROGRAM, SUBROUTINE, BLOCK DATA, or another FUNCTION statement.

Write the FUNCTION statement in the following form:

```
FUNCTION function-name ( dummy-argument-list)
```

FORTRAN determines the type of a function name in the same way as it determines the type of any symbolic name. You can place a type declaration on the same line as the FUNCTION statement as in the following example:

```
CHARACTER*20 FUNCTION employee (dept, empnumber)
```

Or you can declare its type with a separate declaration statement; for example:

```
FUNCTION employee (dept, empnumber)
CHARACTER employee*20
```

If you use a FUNCTION statement to declare a function's type, you cannot declare it again using a type-declaration statement in the same program unit. Note, however, that you must also declare the function's type in each calling program unit, unless it is typed by default.

Assigning a Value to the Function Name

The symbolic function name serves as the main entry point to the function subprogram. The function subprogram must contain a statement that stores a value in the function name. If more than one statement stores a value in the name of the function, the last value stored is the value returned to the calling program.

You can define the function name in any one of the following ways:

- By using an assignment statement
- By using an input statement
- By using the function name as an actual argument in a statement calling another function or subroutine subprogram

In the following example, the function TINTEREST uses an assignment statement to define the value of the function:

```
      FUNCTION tinterest (principal, payment, rate, months)
      REAL interest
      tinterest = 0
      balance = principal
      DO 10 j = 1,months
          interest = balance * (rate/12)
          amount = payment - interest
          balance = balance - amount
          tinterest = tinterest + interest
   10 CONTINUE
      RETURN
      END
```

In the following example, the value returned by the function is the value read by a READ statement:

```
FUNCTION sum(a,b)
READ (5,*) sum
.
RETURN
END
```

In the next example, the value returned by the function is determined when subroutine SUBA returns a value for NUMBER:

```
FUNCTION number(a,b,c)
.
CALL suba(number,x)
.
END
SUBROUTINE suba(j,k)
READ (5,*) k
j = k*3.2
.
END
```

Subroutines

A subroutine subprogram performs a procedure for a calling program unit. The calling program unit can be the main program, a function subprogram, another subroutine, or the subroutine itself.

The subroutine subprogram executes when a CALL statement invokes its name.

Example 4-1 shows the relation between the calling program unit and the subroutine subprogram:

Example 4-1. Calling Program and Subroutine

```
PROGRAM numbers
INTEGER x,y
READ(5,4) x, y
FORMAT (2I2)
IF (x .LT. y) CALL error1              <-- subroutine call
result = x * 100
STOP
END

SUBROUTINE error1                     <-- subroutine name
WRITE (6,1)
1 FORMAT (5x, 'Number is out of range')
RETURN
END
```

When a CALL statement executes, FORTRAN first evaluates any actual arguments that are expressions, and then associates the actual arguments with the dummy arguments specified in the SUBROUTINE statement. The CALL statement then transfers control to the subroutine.

The subroutine subprogram must contain the following statements:

- A SUBROUTINE statement as its first statement
- An END statement as its last statement

In addition to these two statements, you can use any FORTRAN statement in a subroutine except PROGRAM, FUNCTION, BLOCK DATA, or another SUBROUTINE statement. Write the SUBROUTINE statement in the following form:

```
SUBROUTINE name ( dummy-argument-list)
```

As Example 4-1 on page 4-7 shows, a SUBROUTINE statement does not require a dummy argument list or the parentheses that would enclose a dummy argument list.

Subroutines With Alternate Return Specifiers

Normally, when a subroutine executes, it returns control to the statement following the CALL statement. However, you can specify an alternate point of return from a subroutine by including an actual argument in the following form in the actual argument list in the calling statement:

```
* label
```

label is the label of a statement in the calling program unit to which the called subroutine can return.

The following statements declare a subroutine, RATE, in which the third and fourth dummy arguments are the labels of executable statements; and a CALL statement that invokes the RATE subroutine:

```
CALL rate (tonnage, distance, *500, *600)
500 CONTINUE
C -- Arrive here if more than 1000 tons
. . .
600 CONTINUE
C -- Arrive here if less than or equal 1000 tons
END

SUBROUTINE RATE( tons, dist, *, * )
IF (tons .GT. 1000) iexp = 1
iexp = 2
. . .
RETURN iexp
END
```

IEXP must be an integer expression. In the preceding example, if the argument TONS is greater than 1000, RATE transfers control to the statement labeled 500; if TONS is less than or equal to 1000, RATE transfers control to the statement labeled 600. If IEXP is not equal to 1 or 2, RATE returns to the statement following the CALL statement.

Saving Values Computed in Procedure Subprograms

Executing a RETURN or END statement normally terminates the association of actual arguments with dummy arguments in a subroutine or external function subprogram, and causes the subprogram's local variables and arrays to become undefined. However, entities in common blocks and local entities named in DATA statements or SAVE statements remain defined, also after a subprogram returns.

The SAVE statement cannot include dummy argument names, procedure names, or names of entities in common blocks.

If you use a SAVE statement without specifying any names, FORTRAN saves the values of all allowable entities in the subprogram.

Recursion

As an extension to the standard, HP FORTRAN permits recursive calls in subprograms. The following program, which returns the factorial of a number, uses the recursive function FACTORIAL:

```
      INTEGER factorial, j
   10 CONTINUE
      READ (*,*, PROMPT = ' Enter argument: ', END=20) j
      WRITE (*,*) ' Factorial is ', factorial(j)
   20 CONTINUE
      END

      INTEGER FUNCTION factorial (n)
      INTEGER n
      IF (n .GT. 1) THEN
         factorial = n * factorial (n-1)
      ELSE
         factorial = 1
      END IF
      END
```

Using Multiple Entry Points in Functions and Subroutines

The ENTRY statement provides additional entry points for subroutine and function subprograms. The ENTRY statement takes the form:

```
ENTRY name ( argument-list)
```

The ENTRY name, just like the subroutine and function name, must be unique to the executable program. Likewise, the ENTRY argument list must agree in number, type, and length with the actual argument list of the function reference or CALL statement that executes the subprogram beginning with the ENTRY statement. In a function subprogram of type character, every entry point must also be type character.

The following subroutine, HEADING, prints a page heading. The alternate entry point, DETAIL, prints the values of four variables which comprise the data lines of the report.

```
      SUBROUTINE heading (product, price, amount, salesman)
1     WRITE (*,2)
2     FORMAT ('1',T50, 'SALES REPORT FOR MARCH'//T55, 'FREMONT')
      ENTRY detail (product, price, amount, salesman)
4     WRITE (*,5) product, price, amount, salesman
5     FORMAT (T5, I6, T4, F10.2, T4, I5, T4, A)
      END
```

A line count in the calling program determines whether the subroutine executes from statement 1 or statement 4:

```
PROGRAM MAIN
  .
IF (line count .GT. 45) THEN
  CALL heading (prod, cost, amt, sales)
ELSE
  CALL detail (prod, cost, amt, sales)
END IF
  .
END
```

Using Adjustable Dimensions for Arrays and String Variables

If you use an array as an actual argument to a subroutine or function call, the size and type of the associated array in the subroutine or function must correspond to the actual array. The following example uses a subroutine to calculate the average test score for a class of up to one hundred students. The program stores the test scores in the one dimensional array RESULT:

```
      PROGRAM MAIN
      REAL result(100), avg
      READ *, n
      READ *, (result(j), j = 1,n)
      CALL mean(n, result, avg)
      PRINT *, avg
      END

      SUBROUTINE mean(n,data,average)
      REAL data(100), average, sum
      sum = 0
      DO 10 j = 1,n
      sum = sum + data(j)
   10 CONTINUE
      average = sum/n
      END
```

The main program passes values for the number of students and test scores to the subroutine, which accumulates the sum of the test scores, calculates their average, and returns the value of AVERAGE to the calling program. Note that you must dimension the associated dummy array DATA in the subprogram.

The requirement that associated dummy arrays be dimensioned in the subprogram would restrict the availability of the subprogram to calling programs that pass arrays of the same size as declared by the dummy argument. You can use one of the following two methods to avoid explicitly declaring the array size in the subroutine:

- Use an assumed-size array declarator in the subroutine
- Use an adjustable array declarator in the subroutine

Assumed-Size Array Declarator

An assumed-size array declarator takes the form:

array-name(*d* [, *d*]... ,[*lower*:]*)

where *d* is a dimension declarator in the form

[*lower*:] *upper*

You can use an asterisk only for the upper dimension bound of the last dimension.

In the preceding program example, the declaration would be:

```
REAL data(*), average, sum
```

The asterisk indicates that the size of the numeric dummy array is the same as that of the associated numeric actual array.

You can use the * array declarator in associated character arrays, provided the declared element length of the associated arrays is the same.

Adjustable Array Declarator

An adjustable array declarator determines the size of a dummy through the use of a variable dummy argument that indicates a dimension of the array. The following example uses adjustable dimensions to yield the transposition of a matrix:

```
      SUBROUTINE transpose (original, j, k, trans)
      REAL original (j, k), trans (k, j)
      DO 10 m = 1, j
         DO 20 n = 1, k
         trans(n,m) = orig(m,n)
20    CONTINUE
10 CONTINUE
      RETURN
      END
```

You might call TRANSPOSE as follows:

```
REAL matrix(8,9), tmatrix(9,8)
.
CALL transpose (matrix, 8, 9, tmatrix)
```

You must include the adjustable dimension declarator in the dummy argument list.

Assumed-Size Length Declarator

You can also use the asterisk (*) as an assumed-size character length declarator. In the following subroutine, the dummy argument Y assumes the length of the associated actual argument each time MESSAGE is called.

```
SUBROUTINE message(y)
  CHARACTER y*(*)
  PRINT *, y
END
```

If the actual argument is an array name, the length of the associated dummy argument is the length of an array element in the actual argument array.

Using Common Blocks

Common blocks define a common storage area whose contents can be referenced by two or more program units. All subprograms that contain a declaration of the same common block can define and reference the entities included in that common block. The use of common blocks can reduce memory requirements as well as execution time.

You can declare common blocks either as blank common or named common blocks:

```
COMMON name-list                    <-- blank common


COMMON / block-name/ name-list      <-- named common
```

A program can contain more than one named common block, but only one unnamed common block. The list of names in the common statement specifies the variables and arrays that you can access from any program unit that declares that common block.

FORTRAN associates entities in common blocks by storage rather than by name. This means that the order of names in COMMON statements determines which variable names or array element names are associated among program units. In the following example, FORTRAN associates the variable JDEPT in the main program with the variable LOC in the subroutine; it associates SALARY with WAGE, and VACATION with TIME OFF:

```
PROGRAM MAIN
COMMON /employee/jdept, salary, vacation


SUBROUTINE overtime
COMMON /employee/loc, wage, time off
```

Considerations

- Associated entities in common blocks must correspond with respect to type.
- If a common block contains a character variable or character array, all names in that block must store character values.
- COMMON statements are nonexecutable and must precede all DATA and executable statements in the program unit.
- HP FORTRAN allows a common block to have the same name as a program unit. The ANSI standard does not permit this.

For additional information on using common blocks and memory organization, see Section 12, Memory Organization.

The Block Data Subprogram

The BLOCK DATA subprogram is a nonexecutable declaration subprogram that assigns initial values to entities in common blocks. By using the BLOCK DATA subprogram to initialize common block entities you can save the execution time and code space that would otherwise be needed to initialize these entities.

A BLOCK DATA subprogram can be either named or unnamed. A FORTRAN program, however, can contain only one unnamed BLOCK DATA subprogram.

The BLOCK DATA subprogram has the following form:

```
BLOCK DATA [ name ]
.
END
```

ABLOCK DATA subprogram can contain only the following statements:

type declaration	COMMON
IMPLICIT	SAVE
PARAMETER	DATA
DIMENSION	EQUIVALENCE

Considerations

- If the BLOCK DATA subprogram contains an IMPLICIT statement, it must appear after the BLOCK DATA statement but before any other statements in the subprogram.
- DATA statements must appear before the END statement but after all other statements.
- A COMMON statement must include all elements of the common block, even if you initialize only some of these elements in the BLOCK DATA subprogram.
- Although you can initialize entities in several common blocks in one BLOCK DATA subprogram, you can use only one BLOCK DATA subprogram to initialize data in one particular common block.
- A BLOCK DATA subprogram cannot contain local variables or arrays. Local variables and arrays must be in common blocks.
- BLOCK DATA subprograms cannot contain RECORD declarations, whether they are in common blocks or not.

The following sample program specifies that the variables COMMISSION, TAX, SHIPPING, and VAT are in the named common block COST, and that the array QUANTITY is in the named common block PRODUCT:

```
BLOCK DATA cost data INTEGER*4 quantity
  REAL commission, tax, shipping
  COMMON /product/quantity(1000)
  COMMON /cost/commission, tax, shipping, vat
  DATA commission, tax, shipping/.20, .06, .15/
  DATA quantity/1000 * 0/
END
```

The first DATA statement initializes three of the variables in COST to the values shown. The second DATA statement initializes each element in the QUANTITY array to zero.

5

Introduction to File I/O in the HP NonStop Environment

This section introduces FORTRAN I/O in the HP NonStop environment. Topics covered in this section include:

FORTRAN I/O Statements

FORTRAN I/O statements include data transfer statements, file status statements, and file positioning statements. Table 5-1 summarizes FORTRAN I/O statements. For a detailed description of each statement, see Section 7, Statements.

Data transfer statements direct the transfer of data between internal storage and external media or internal files. Data transfer statements include:

- Formatted READ, WRITE, and PRINT statements
- Unformatted READ and WRITE statements
- List-directed READ, WRITE, and PRINT statements

File status statements, OPEN, CLOSE, and INQUIRE, determine the properties of the connection to the external medium.

Table 5-1. FORTRAN I/O Statements (page 1 of 2)

Name	Action
BACKSPACE	Positions file just before the preceding record.
CLOSE	Disconnects file from unit.
ENDFILE	Writes an end of file record as the next record of a file.

Table 5-1. FORTRAN I/O Statements (page 2 of 2)

Name	Action
INQUIRE	Ascertains properties of a file or of its connection.
OPEN	Connects a structured file to a unit. Creates an unstructured file and connects it to a unit.
POSITION	Enables random access of files.
PRINT	Outputs data to the preconnected output unit, unit 6.
READ	Inputs data from specified unit or file.
REWIND	Positions connected file to its initial point.
WRITE	Outputs data to a specified unit.

Character literals used as control specifiers in I/O statements (for example, 'DIRECT', 'EXACT', 'FORMATTED', and so forth) must be in uppercase characters.

The file positioning statements, REWIND, BACKSPACE, ENDFILE, and POSITION, determine the position of a file.

File status and positioning statements are also known as auxiliary I/O statements.

Records

Input and output involve reading records from files or writing records to files. A record is a sequence of values or characters. FORTRAN uses three kinds of records: formatted records, unformatted records, and end-of-file records.

A formatted record consists of a sequence of ASCII characters. The length of a formatted record depends primarily on the number of characters put into the record when it is written. A record can have a length of zero. You can read formatted records with either formatted or unformatted I/O statements.

An unformatted record consists of a sequence of values and can contain character data, noncharacter data, or no data. The length of an unformatted record is measured in bytes and depends on the output list used when you write the record. You can transmit unformatted records using either formatted or unformatted I/O statements.

You can write an end-of-file record only as the last record of a file which you can access sequentially. The end-of-file record does not have a length attribute.

Note. Records used for I/O operations are not the same as records defined by RECORD and END RECORD declaration statements. This text refers to the former as records and to the latter as RECORDs.

FORTRAN Files

A file is a sequence of records. When a program executes, those files that are available to it are said to exist for that program.

A file that exists for a program might not contain any records, as would be the case with a newly created file. A file can be known to the Guardian file system but, for security reasons, might not be visible to a program. The INQUIRE, OPEN, CLOSE, WRITE, and PRINT statements can refer to files that do not exist.

Table 5-2. File Attributes

Attribute	Definition
Name	Every file has a name that is defined either by the user when the file is created or by the system when the file is opened. for additional information, see HP File Names on page 5-5.
Position	A file connected to a unit has a position attribute. The execution of a WRITE, READ, PRINT, BACKSPACE, REWIND, ENDFILE, OPEN, or POSITION statement affects the file's position.
Initial Point	The initial point of a file is the position just before the first record.
Terminal Point	The terminal point of a file is the position just after the last record.
Current Record	The current record is the record where the file is currently positioned.
Preceding Record	If the current record is n, the preceding record is the n-1 record.
Next Record	If the current record is n, the next record is the n+1 record.

External and Internal Files

A FORTRAN file is either external or internal.

An external file is a collection of records stored on a medium external to primary memory. An internal file is a means of transferring data from one location to another within memory.

The attributes of external files are determined by the device that handles their data transfer, the file structure, and the method you use to access the file. The subsection File Characteristics on page 5-16 describes the file structure of HP external files.

An internal file is a character variable, character array, character array element, RECORD name, or character substring. It has the following attributes:

- FORTRAN treats a file that is a character variable, character array element, substring, undimensioned RECORD, or a RECORD array element as a single record. The length of the record equals the length of the variable, array element, substring, undimensioned RECORD, or RECORD array element. FORTRAN treats RECORDs like character variables and arrays.

- FORTRAN treats each element of a file that is a character array or a RECORD array as a record. The ordering of the records is the same as the ordering of the elements of the array. The length of a record equals the length of an array element.
- A variable, array element, substring, RECORD, or RECORD array element that is a record of an internal file is defined when you write that record. If the number of characters written is less than the declared length of the record, FORTRAN pads the remainder of the record with blanks.
- A record can be read only if it has been defined.
- An internal file is always positioned at the beginning of the first record prior to data transfer.

Considerations

- You can access internal files only sequentially.
- You cannot use auxiliary I/O statements to refer to internal files.

File Properties

To access an external file that exists for a program, you must connect the file to a FORTRAN unit. At the time of connection a file has the following properties:

- A unit number (see Units on page 5-8)
- A file name
- A form of data (formatted, unformatted, or both)
- A method of access (sequential, direct, or keyed)
- A record length

Table 5-3 lists the default values for file attributes that you do not specify when connecting the file to a FORTRAN unit.

Table 5-3. FORTRAN Default File Attributes

Attribute	Setting
File name	A temporary file in the default disk volume
File type	Unstructured
File code	0
Extent size	One page (2048 bytes)
Record size	132 bytes
Access mode	'I-O'
Exclusion mode	'SHARED'
Tandem File	Names

HP File Names

You can specify a file name when you connect a file to a unit as in the following example:

```
OPEN (15, FILE='newaccts', FORM='FORMATTED', STATUS='NEW')
```

A file name specifies the location of a file and consists of four 8-character names separated by periods. A file name consists of a:

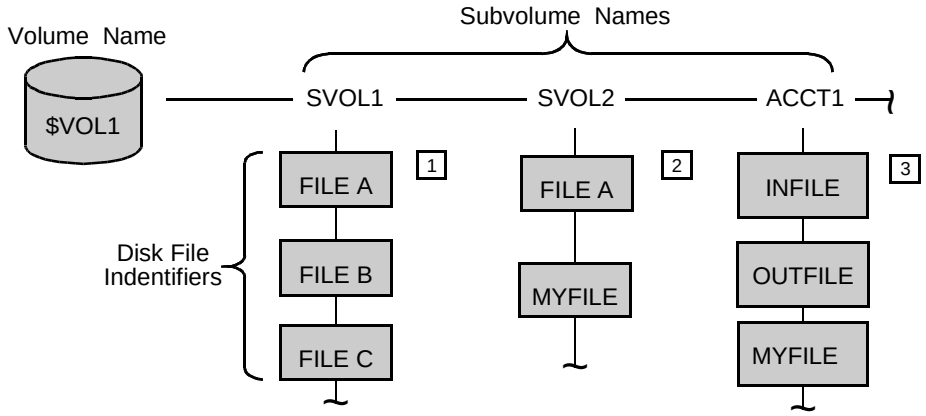
- Node name (system name)
- Volume name
- Subvolume name
- File ID

Here is an example of a fully qualified file name:

```
\ANODE.$AVOLUME.MYSUB.MYFILE
```

Figure 5-1 illustrates the preceding disk file name model.

Figure 5-1. Disk File Organization



In a network, the file name can also include a node name, as in the following example:

```
\PARIS.$SALES.JONES.PROGRAMS
```

The preceding file name identifies the file identifier PROGRAMS, in the subvolume JONES, in the volume $SALES, at node \PARIS. As shown in the preceding example, you must insert a period between file name components.

Considerations

- The first character of the node name must be a backslash (\).
- The first character of the volume name must be a dollar sign ($).
- Each component of a file name can contain up to eight characters.
- If ENV OLD is in effect, the volume name in a file name that references a file at a remote node can be a maximum of seven characters including the dollar sign. If ENV COMMON is in effect, the VOLUME name can be eight characters including the dollar sign.

Formatted and Unformatted Files

An unformatted file consists of unformatted records, and a formatted file consists of formatted records. FORTRAN processes unformatted files more quickly than it processes formatted files, because the processing of unformatted files does not require that data be converted to an external representation. Note, however, that the HP implementation of FORTRAN permits a file to contain both formatted and unformatted records.

You specify the data format for a file when you connect the file to a unit, as in the following example:

```
OPEN (22, FILE='budget', FORM='FORMATTED')
```

File Access

You can use three methods for accessing the records of an external file: sequential, direct, and keyed.

When you access a file sequentially, you access each record in the file in the order in which the records appear in the file. You can access sequential files only by sequentially reading or writing each record in the file. If you need to read the tenth record of a sequential file, you must process the preceding nine records first. You can only access internal files sequentially.

You can use direct access I/O to write or read specific records in a file without accessing any other record in the file. You can access an external file either sequentially or directly.

If you use one of the three file types described in Structured Files on page 5-18, you can also use primary or alternate key fields for keyed access when you read the file.

You determine the access method for an external file when you connect that file to a unit; for example:

```
OPEN (100, FILE='oldaccts', ACCESS='DIRECT')
```

Some files are restricted to only one access method. For example, you can access printer files only sequentially. Table 5-4 on page 5-7 shows the valid access methods for HP-defined files.

When connected for sequential access, a file has the following attributes:

- If direct access is not permitted for the file, record order corresponds to the order in which the records were written. If you can also access the file directly, the order of the records is the same as that specified for direct access.
- The file's records can be formatted, unformatted, or mixed. This is an HP extension.
- The last record can be an end-of-file record.

When connected for direct access, a file has the following attributes:

- The end-of-file record (if any) is not considered to be a part of the file.
- The file's records can be formatted, unformatted or mixed. This is an HP extension.
- Each record of the file has a unique, nonnegative record number that you must specify when you write the record. You can never change the number of a record.
- You cannot delete the record, but you can rewrite it.
- In an EDIT format file, a record number is an EDIT line number times 1000 (that is, line number 12.34 is record number 12340). A line can have record number 0. In other unstructured files and in relative files, record numbers are positive integers.
- You can update (rewrite) records in all file types capable of direct access.
- You can delete records in some file types by rewriting a zero-length record with an unformatted WRITE statement that specifies UPDATE = .TRUE. and does not have a data list.
- Records are ordered by record number, but you can read or write records in any order. For example, you can write record 3 although records 1 and 2 do not exist, or you can read any defined record while the file is connected to a unit.

The attributes of files that can be connected for keyed access are described in Structured Files on page 5-18.

Table 5-4. FORTRAN Access Methods for HP-defined Files (page 1 of 2)

	Access Method		
File Type	**Sequential**	**Direct**	**Keyed**
Magnetic tape	Y	N	N
Card reader	Y	N	N
Printer	Y	N	N
Terminal	Y	N	N
Process	Y	N	N
$RECEIVE	Y	N	N
EDIT format file	Y	Y	N

Table 5-4. FORTRAN Access Methods for HP-defined Files (page 2 of 2)

	Access Method		
File Type	**Sequential**	**Direct**	**Keyed**
Unstructured	Y	Y	N
Entry-sequenced	Y	N	Y
Key-sequenced	Y	N	Y
Relative	Y	Y	Y

Logical Record Length

The HP Formatter software package performs all FORTRAN I/O, both formatted and unformatted. The Formatter allocates a buffer whose size is equal to the record size you specify with one of the following:

- The RECL specifier in an OPEN statement.
- The REC specifier in a UNIT compiler directive.
- The REC specifier in a TACL ASSIGN command.
- The REC specifier in a FUP CREATE command that creates the file (for structured files only).

If you use more than one method to specify the size of a record, the record length specified in an OPEN statement overrides the record length specified in an ASSIGN statement, which overrides the record length specified in a UNIT directive. For a structured file, the REC specifier in the FUP CREATE command overrides all other record-length specifications.

If you open a file without specifying its record length by any of the above methods, FORTRAN uses the configured record length for the device the file is on (for example, 80 bytes for a card reader). For devices such as disk volumes and processes, FORTRAN uses 132-byte records.

The following statement connects the file INFO to unit 49, and specifies the record length to be 124 characters:

```
OPEN (49, FILE='info', RECL=124)
```

Units

A FORTRAN program accesses an external file by “connecting” the file to a “unit” for that program. Units are the handles by which FORTRAN I/O statements refer to files after they are connected. Files have names, but units have numbers. Valid unit numbers are integers in the range 1 to 999 inclusive. The OPEN and INQUIRE statements are the only FORTRAN I/O statements that can refer to a file by its file name. All other statements must use unit numbers.

For any Guardian program running under the operating system, a file can:

- Exist or not exist
- Be open or not open

For a FORTRAN executable program, a unit can:

- Exist or not exist
- Be assigned or not assigned
- Be connected or not connected

File Existence

A disk file exists if it is known to the system. It need not have any data records in it.

If you do not name the file at the time of connection, FORTRAN creates a temporary file, which it purges when you close the file or your process terminates.

You create a structured file only by using utility programs (such as FUP) that initialize the required record key indexing data structures within the file. For additional information, see Structured Files on page 5-18.

A nondisk file name is simply a device name or a process name. Such a file exists if and only if the device or process exists.

Opening a File

A file is open for an executing program if the system is prepared to accept requests from that program for transferring data to or from that file, for example, if the program has at least one Guardian open file number for the file. The open state begins and ends with calls to the Guardian file-system procedures OPEN and CLOSE, respectively. The FORTRAN processor makes such calls automatically when a FORTRAN unit is connected or disconnected. A file must exist before it can be opened. For a disk file, FORTRAN automatically creates the file, if necessary, to satisfy this requirement. Thus, for FORTRAN, creating a file and opening a file can be one single operation.

A disk file can be opened by more than one opener at the same time. The multiple openers can be in different processes or in the same process. The openers can be from routines written in FORTRAN or in other HP NonStop languages. FORTRAN associates each open of a Guardian file with the unit number you specify in a FORTRAN OPEN statement. Thus, you can have one or more unit numbers associated with the same Guardian file, each with its own Guardian file number.

Unit Existence

A unit exists for a FORTRAN executable program if and only if its unit number has been made known in one or more of the following ways:

- The unit number appears, in the form of an integer constant or a symbolic constant whose value is an integer, as the unit number parameter in at least one FORTRAN I/O statement, of any kind, anywhere in the compilation.
- The unit number is defined in a UNIT compiler directive anywhere in the compilation. Use a UNIT compiler directive to establish the existence of a unit only if all the I/O statements that refer to that unit use a variable or an expression (other than a constant) as their unit designator.
- Units 4, 5, and 6 always exist by default.

FORTRAN I/O statements other than CLOSE or INQUIRE must refer to units that exist for the program.

Unit Assignment

A unit is assigned for a FORTRAN executable program if and only if the unit number has been associated with a file name. Assignment can be established in any of the following ways:

- Units 4, 5, and 6 are always assigned by default. Unless one of the other methods listed below specifies otherwise, units 4, 5, and 6 are automatically assigned to the program's home terminal, standard input file, and standard output file, respectively. The file names for these files are specified or assumed in the runoptions TERM, IN, and OUT, respectively, in the command that initiates execution of the program.
- When the program is compiled, the UNIT compiler directive can specify the file name for the unit. If the unit number is 5 or 6 this file-name specification overrides the default described above. If the file name is not fully qualified, execution-time defaults are supplied for the missing parts of the file name. The UNIT directive can also specify a unit name for use in ASSIGN commands. For more information, see The ASSIGN Command on page 5-11.
- Before you run your program, you can specify the file name for the unit in a TACL ASSIGN command. The file name you specify in the ASSIGN command overrides the file name specified or assumed at compile time by either of the preceding methods. The ASSIGN command associates a logical name, for example, FT *uuu* or the name you specify in a UNIT directive, with a physical—or external—file name, as shown in the following:

  ```
  ASSIGN FT009, external-name
  ```

- During execution of the program, the FILE specifier of an OPEN statement can establish a new file name for the unit, overriding all the above methods.

Note that assigning a file name to a unit number does not, by itself, cause the file to exist (that is, the assignment does not create the file).

The ASSIGN Command

The TACL ASSIGN command enables you to assign the name of an actual file to a unit specified in a program and to specify the characteristics of the file at run time.

```
ASSIGN [ logical-unit [,[ filename ][, create-spec]... ] ]
```

logical-unit

is the unit name specified in the UNITNAME option of the UNIT compiler directive. The name can be from 1 to 31 characters including letters, digits, and hyphens. If no UNIT directive specifies the name of a unit, the default unit name has the form FT *uuu*, where *uuu* is a three-digit decimal integer from 1 through 999 which specifies the unit to be connected: Unit 2 is FT002, unit 99 is FT099, and so forth.

filename

is an HP file name (\ *node.$ volume. subvol. fileid*). An OPEN statement can override this value with a FILE specifier.

create-spec

is a specification for one or more of the following file attributes:

exclusion-spec

is the exclusion mode of *logical-unit* and determines how other processes can access the file. It is one of the following keywords:

Keyword	Meaning
EXCLUSIVE	No other processes can access the filename while the program containing *logical-unit* has the file open.
SHARED	Other processes can read and write to filename while the program containing *logical-unit* has the file open. This is the default value.
PROTECTED	Other processes can read, but not write to, filename while the program containing *logical-unit* has the file open for write access.

An OPEN statement can override the value of *exclusion-spec* with a PROTECT specifier.

access-spec

is one of the following access modes for *logical-unit*.

Keyword	Meaning
I-O	This process can both read from and write to the file. I-O is the default value.
INPUT	This process can only read the file.
OUTPUT	This process can only write to the file.

An OPEN statement can override the value of *access-spec* with the MODE specifier.

REC *record-size*

is an integer ranging from 1 through 65,535 which sets the length (in bytes) of records in *logical-unit*.

An OPEN statement can override this value with a RECL specifier.

CODE *file-code*

specifies the file code of the file being connected. 101 specifies an EDIT format file.

extent-spec

assumes one of the following three forms:

```
EXT pri-extent
EXT ( pri-extent )
EXT ( [ pri-extent ] , sec-extent )
```

where *pri-extent* is an integer in the range from 1 to 65,535 inclusive that specifies the size of the primary extent, and *sec-extent* is an integer in the same range that specifies the size of each secondary extent. Each extent size is the number of 2048-byte disk pages to be allocated at one time.

BLOCK *block-size*

is an integer in the range of 1 through 65,535 that specifies the size (in bytes) of the data blocks used by *logical-unit*. Although you can specify the block size, the FORTRAN (and COBOL85) run-time environments do not use the value you specify.

For example, the following TACL ASSIGN command assigns the file name DATAFILE to unit 2:

```
1> ASSIGN FT002, datafile, input, exclusive
```

To get a list of the attributes of a logical file, enter:

```
1> ASSIGN logical-unit
```

If you entered FT002 for logical-unit, the following information would be displayed:

```
FT002
      PHYSICAL FILE: DATAFILE
      EXCLUSION: EXCLUSIVE
      ACCESS: INPUT
```

To get a list of the assigned attributes of all logical units, enter:

```
1> ASSIGN
```

For additional information on the ASSIGN command, see the *TACL Reference Manual*.

Unit Connection

A BACKSPACE, ENDFILE, OPEN, POSITION, PRINT, READ, REWIND, or WRITE statement establishes a connection to a unit. A CLOSE statement closes the connection to a unit. An INQUIRE statement does not alter whether a unit is connected or not.

A unit cannot be connected to more than one file at a time, but a file can be connected to any number of units at the same time.

No FORTRAN I/O statement requires that the unit be connected before you execute the statement. All I/O statements except CLOSE and INQUIRE automatically connect the unit if it is not already connected. That is, they create the file if it does not already exist (except for structured files), and open it if necessary.

If an OPEN statement opens a unit that is already connected, the unit is closed and opened anew. For example, the effect of the following statements is the same as if a CLOSE statement without a STATUS specifier had been executed between the opens for FILE1:

```
OPEN (15, FILE= 'file1')
OPEN (15, FILE= 'file2')
```

If any I/O statement connects a unit that is not assigned to a file name (the unit number is not 4, 5, or 6, and no UNIT directive or ASSIGN command has specified a file name for the unit, and the I/O statement is not an OPEN statement with FILE = a nonblank name), then the system creates a new temporary file with a unique file name on the default disk volume for the program, and opens it, thus connecting the new file to the unit.

Specifying File Attributes

You can use the UNIT directive and the ASSIGN command to specify other file attributes in addition to the file name. For details, see the UNIT Compiler Directive on page 10-67 and the description of the ASSIGN command in this section.

If you use more than one method to specify the same file attribute, the compiler uses the following order of precedence to determine the attribute value:

```
OPEN statement                <-- highest
ASSIGN TACL command
UNIT compiler directive
Default                       <-- lowest
```

That is, an attribute value specified in an OPEN statement overrides a different value specified for the same attribute of that unit in an ASSIGN command, and so forth.

Table 5-5 shows the different ways that you can specify file attributes. Note that you cannot specify the file type, file code, extent sizes, or block size in an OPEN statement, and that you can specify structured disk file types (relative, entry-sequenced, and keysequenced) only by creating the file outside of FORTRAN, for example by using the CREATE command in the FUP utility program.

If an executing FORTRAN program opens a file that already exists (that is, the file has been created, though it might be empty), any attributes specified for the file in UNIT directives, ASSIGN commands, or the OPEN statement, must agree with those of the actual file.

If the file does not already exist, FORTRAN creates the file with the attributes you specify.

Table 5-5. File Attribute Specification (page 1 of 2)

Attribute	Default	Unit Directive	Assign Command	Open Statement
File name	*temporary*	*file name*	*file name*	FILE = **cexp**
File type	Unstructured	----	----	----
File code	0	CODE *num*	CODE *num*	----
Extent Size	1 page	EXT (*pri*, *sec*)	EXT (*pri*, *sec*)	----
Record size	132	REC *num*	REC *num*	RECL = *exp*
Access mode	I-O	*access*	*access*	MODE = *cexp*
Exclusion	SHARED	*exclusion*	*exclusion*	PROTECT = *cexp*
Unit name	FT*uuu*	UNITNAME *name*	*logical unit*	----

temporary	is a unique file name created by the system.
file name	is an HP file name. If it is not fully qualified, run-time defaults are supplied for the missing parts of the file name. In the OPEN statement, *cexp* is a character expression whose value is in the file name.

Table 5-5. File Attribute Specification (page 2 of 2)

Attribute	Default	Unit Directive	Assign Command	Open Statement
num	is an unsigned integer. In an OPEN statement, *exp* is an integer expression.			
access	is I-O, INPUT, or OUTPUT. In an OPEN statement, *cexp* is a character expression whose value is one of these.			
exclusion	is SHARED, PROTECTED, or EXCLUSIVE. In the OPEN statement, *cexp* is a character expression whose value is one of these.			

Example: Connecting Units to Files

As an example, consider the following FORTRAN program:

```
?UNIT (2, input.data, REC 200, UNITNAME vector)
?UNIT (3, result.file, REC 200, EXCLUSIVE)
?UNIT (6, , CODE 101)
   PROGRAM example
   CHARACTER result file * 34, text * 72
   REAL values (100)
   READ (4,*, PROMPT =' Iterations: ', END = 30)
   & iterations
   READ (4, 10, PROMPT = ' Result file: ', END = 30)
   & result file
   READ (5, 10) text
10 FORMAT (A)
   WRITE (6, 20) text
20 FORMAT ('1', A)
   WRITE (1) iterations
   READ (2) values
   ...
   OPEN (3, FILE = result file, RECL = 400)
   WRITE (3) values
30 listing = 7
   WRITE (listing, 20) text
   STOP
   END
```

After you compile the program but before you run it, you might enter the following ASSIGN commands:

```
1> ASSIGN vector, $data.test.input, REC 400
2> ASSIGN ft003, , OUTPUT, EXT 4
```

When the object program runs, units 4, 5, and 6 are automatically connected to the home terminal, standard input file, and standard output file respectively. If the standard output file does not already exist, FORTRAN creates it as an EDIT format file, because the UNIT 6 directive specifies file code 101.

The first reference to unit 1 is a WRITE statement. FORTRAN creates the file (as an unstructured code 0 disk file in the default volume with a temporary file name, onepage primary and secondary extent sizes, and 132-byte block and record lengths) and opens it, all automatically, using defaults in the absence of a UNIT directive, ASSIGN command, or OPEN statement for the unit.

Unit 2 is specified in both a UNIT directive and an ASSIGN command, using the unit name VECTOR to establish this linkage. The file name and record length in the ASSIGN command override those of the UNIT directive. The record length provides for 100 type REAL elements of 4 bytes each.

Unit 3 is also specified in both a UNIT directive and an ASSIGN command. In this case, the default unit name FT003 (corresponding to unit number 3) provides the linkage. The UNIT directive specifies a file name, a record length, and a protection attribute, while the ASSIGN command specifies an access mode and an extent size.

The OPEN statement specifies a file name and a record length, overriding those of the UNIT directive. The typed-in file name will be connected.

The program’s execution will terminate abnormally when it gets to the WRITE statement after statement 30, because unit 7 does not exist.

File Characteristics

This subsection describes how you use FORTRAN files in the NonStop environment, in which files are either structured or unstructured. An unstructured file is a byte array. It is normally used as a code file or EDIT format file. The application process determines the length and locations of records within the file. Structured files contain logical records whose order depends on the type of structured file used.

Unstructured Files

You address data stored in an unstructured file (except for EDIT format files) in terms of fixed-length numbered records. The first record in a file is record number 1, the second is record number 2, and so forth. All records in the file have the same length and are stored without any record delimiters, data compression, record indexing, or other data structuring of any kind. The total size of the file (in bytes) is the record length times the number of records in the file (given by the highest-numbered record written so far). Thus, in an unstructured file:

- All physical records are the same length.
- All logical records are the same length.
- You can access records sequentially or directly.
- You can read, write, or update any record in the file but you can never delete a record once it has been written.
- You cannot use alternate keys.

EDIT Format Files

EDIT format files are unstructured files. Each line of an EDIT format file is a record of that file. The record number equals the EDIT line number times 1000; for example, line number 12.34 is record number 12340.

You can create an EDIT format file in any of the following ways:

- Specify CODE 101 in a UNIT directive
- Specify CODE 101 in a TACL ASSIGN command
- Use the FUP utility. The following commands create the EDIT format file CREDITOR:

```
1> FUP
File Utility Program - T6553D10 - (08JUN92) System \ASYS
Copyright Tandem Computers Incorporated 1981, 1983, 1985-1992
-set type u
-set code 101
-create creditor
CREATED - $AVOL.USER.CREDITOR
-exit
2>
```

In an EDIT format file:

- Physical records vary in length. The maximum length is always 239 bytes.
- Logical records also vary in length, up to the maximum length specified when the file is connected.
- You can read records sequentially.
- You can rewind the file.
- You cannot use keyed access.
- For a FORTRAN program running as a NonStop process pair:
  - You cannot write, update, or delete records in the file.
  - You cannot backspace over records in the file.
  - You cannot use direct access.
- For a FORTRAN program that is not running as a NonStop process pair:

- You can read and write records sequentially.
- You can backspace the file, rewind the file, or write an endfile record to the file.
- You can use direct access for reading and writing records in the file. The record number is the record's EDIT line number times 1000. Use values of -1 for the beginning of the file and -2 for the end of the file.
- You can change the length of a record as long as the length does not exceed the specified record size for the file.
- You can delete records.

- If you READ the file directly with REC= specifying the nonnegative record number (including 0) of a record that does not exist, then:
  - If the specified record number is beyond the end of the file, you get an end-offile indication.
  - If the specified record number is not beyond the end of the file, you get the record with the smallest record number that is greater than or equal to the specified record number. FORTRAN does not consider this an error. Consequently, you should have an INQUIRE statement with a NEXTREC specifier after every such READ, to determine the record number of the record actually obtained.

Note. EDIT format files are not structured files and are not protected by TMF. Although you can do random positioning and updating of EDIT format files in FORTRAN programs (except in programs that run as NonStop process pairs), you should avoid such actions for applications in which fault tolerance or data integrity is important.

Structured Files

Structured files are entry-sequenced, relative, or key-sequenced files. You create a structured file using the File Utility Program (FUP). You cannot create a structured file using FORTRAN.

Use the following procedure to create a structured file:

1. Start the File Utility Program by entering the following at the TACL prompt:

   ```
   1> FUP
   ```

2. Assign values to file creation parameters with the SET command. FUP maintains a table of creation parameters. The values in this table determine the attributes of any file you create with FUP. FUP uses the following codes to identify file types:

   ```
   Entry-sequenced    E
   Relative           R
   Key-sequenced      K
   ```

3. Check the values of file creation parameters with the SHOW command. Note that the SHOW command displays default values for file attributes in addition to the values you specify.
4. Create the file with the CREATE command. FUP checks the values you chose with the SET command and creates a file if those values will result in a legal file.
5. Specify STATUS='OLD' in the OPEN statement for a structured file. This is the default when the file already exists.
6. To delete a record from a structured file do an unformatted WRITE to the file. Specify UPDATE=.TRUE. but do not include a data list on the WRITE statement.

For a complete description of structured files see the *ENSCRIBE Programmer's Guide*.

Entry-Sequenced Files

An entry-sequenced file appends records to the end of a file in the order in which they are written. In an entry-sequenced file:

- Records are searched sequentially from the beginning of the file.
- The length of a record depends upon the length specified when it is written. Once you delete a record, its space can only be used for another record of the same size.
- Records added to the file can vary in length, but once a new record is added, its length cannot change.
- Records in the file can be updated but not deleted.
- You cannot use direct access.
- You can access records by alternate keys. See Using Alternate Keys on page 5-22.
- Records are always written at the end of the file, also after a REWIND or OPEN statement. The only exception to this is a WRITE statement with an UPDATE = .TRUE. specifier.

The following example shows how to create the entry-sequenced file VISITORS. The default record size for the file is 80 bytes.

```
1> FUP
File Utility Program - T6553D10 - (08JUN92) System \ASYS
Copyright Tandem Computers Incorporated 1981, 1983, 1985-1992
-SET TYPE E                          <-- Set file type
-SHOW                                <-- Show current values
   TYPE E
   EXT ( 1 PAGES, 1 PAGES )
  REC 80
  BLOCK 4096
  MAXEXTENTS 16
-CREATE visitors                     <-- Create the file
CREATED - $JUICE.BUJES.VISITORS
-EXIT                                <-- Exit FUP
2>
```

Your FORTRAN program might include the following statement to connect the file to a unit:

```
OPEN (13, FILE='visitors', ACCESS='SEQUENTIAL',
&     STATUS='OLD', FORM='FORMATTED')
```

The following statement adds a record to the file:

```
WRITE (13, FMT=20) name, address, date
```

The following statements update a record in the file:

```
READ (13, FMT=30, REC=80, UPDATE=.TRUE.) name, address, date
address='145 N. Main'
WRITE (13, FMT=30, UPDATE=.TRUE.) name, address, date
```

Relative Files

A relative file stores records relative to the beginning of the file, according to a record number supplied by the application program. In a relative file:

- All physical records are the same length.
- Logical records can vary in length.
- Each record is uniquely identified by a record number, which denotes an ordinal position in a file. You can access a record by record number or by alternate key. (For more information about alternate key access, see Using Alternate Keys on page 5-22.)
- You can delete a record.
- You can change the length of a record so long as the length does not exceed the specified record size for the file.
- A value of -2 for the REC= specifier inserts a record in the first empty position. A value of -1 appends a record to the end of the file.
- You can access records sequentially or directly.

- If you READ the file directly with REC= specifying the nonnegative record number (including 0) of a record that does not exist, then
  - If the specified record number is beyond the end of the file, you get an end-offile indication.
  - Otherwise, you get the record with the smallest record number that is greater than or equal to the specified record number. FORTRAN does not consider this an error. Consequently, you should execute an INQUIRE statement with the NEXTREC specifier after this type of READ.

The following example shows how to create the file RESERVED:

```
1> FUP
File Utility Program - T6553D10 - (08JUN92) System \ASYS
Copyright Tandem Computers Incorporated 1981, 1983, 1985-1992
-SET TYPE R                       <-- Set file type
-SET REC 120                      <-- Set record length
-SHOW                             <-- Show the current values
    TYPE R
    EXT (1 PAGES, 1 PAGES)
    REC 120
    BLOCK 4096
    MAXEXTENTS 16
-CREATE reserved                <-- Create file
CREATED - $JUICE.BUJES.RESERVED
-EXIT                           <-- Exit FUP
2>
```

To connect the file to a unit for direct access, your program might contain the following:

```
OPEN (15, file='reserved', ACCESS='DIRECT',
&     FORM='FORMATTED', STATUS='OLD')
```

To read a record from the file:

```
READ *, k
READ (15, FMT=20, REC=k) name, date, flight
```

To update a record:

```
READ *, k
WRITE (15, FMT=20, UPDATE=.TRUE., REC=k) name, date, flight
```

Key-Sequenced Files

Records in key-sequenced files are logically stored in order of ascending primary-key values. A primary key is a record field that uniquely identifies the record. You can also use alternate keys to access data in a key-sequenced file.

In a key-sequenced file:

- The space occupied by the record depends on the length you specify when you write it.
- You can shorten, lengthen, or delete records.

The following example shows how to create the key-sequenced file VENDORS.

```
1> FUP
File Utility Program - T6553D10 - (08JUN92) System \ASYS
Copyright Tandem Computers Incorporated 1981, 1983, 1985-1992
-SET TYPE K                          <-- Set file type
-SET REC 120                         <-- Set record length
-SET KEYLEN 24                       <-- Set primary key length
-SHOW <-- Show values
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 120
    BLOCK 4096
    IBLOCK 4096
    KEYLEN 24
    KEYOFF 0
    MAXEXTENTS 16
-CREATE vendors                    <-- Create file
CREATED - $JUICE.BUJES.VENDORS
-EXIT                              <-- Exit FUP
2>
```

The following statement opens the VENDORS file:

```
OPEN(22, file='vendors', STATUS= 'OLD', ACCESS='SEQUENTIAL',
&    FORM='FORMATTED')
```

The following statement writes a record to the file:

```
WRITE(22, FMT=30) name, address, amount, discount
```

You can position the file for a subsequent read or write:

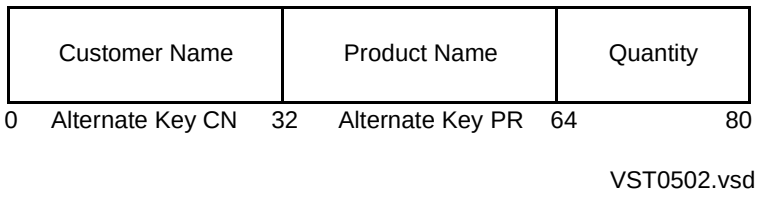
```
READ *, name
POSITION(22, KEY=name, MODE='EXACT')
```

Using Alternate Keys

To use alternate keys to access data in a structured file, you must use FUP SET commands to:

- Define a two-letter key specifier that identifies the alternate key as an access path for positioning the file.
- Define the record offset where the alternate-key field begins.
- Define the length of the alternate key field.
- Create an alternate-key file for each structured file having one or more alternate keys.

The following record in the relative file SALES can be accessed by alternate keys:



VST0502.vsd

The following FUP commands create the relative file SALES:

```
1> FUP
File Utility Program - T6553D10 - (08JUN92) System \ASYS
Copyright Tandem Computers Incorporated 1981, 1983, 1985-1992
-SET TYPE R
-SET REC 80
-SET ALTKEY ("CN", KEYOFF 0, KEYLEN 32)  <-- Define alternate
-SET ALTKEY ("PR", KEYOFF 32, KEYLEN 32)     keys
-SET ALTFILE (0, altsale)                <-- Name alternate-
-SHOW                                        key file
    TYPE R
    EXT (1 PAGES, 1 PAGES)
    REC 80
    BLOCK 4096
    ALTKEY ("CN", FILE 0, KEYOFF 0, KEYLEN 32)
    ALTKEY ("PR", FILE 0, KEYOFF 32, KEYLEN 32)
    ALTFILE (0, $JUICE.FTRAN.ALTSALE)
    ALTCREATE
    MAXEXTENTS 16
-CREATE sales
CREATED - $JUICE.FTRAN.SALES
CREATED - $JUICE.FTRAN.ALTSALES
-RESET
-EXIT
2>
```

To connect the file to a unit for sequential access, your program might contain the following:

```
OPEN (20, file='sales', ACCESS='SEQUENTIAL', RECL=80,
&     FORM='FORMATTED', STATUS='OLD')
```

The following statements position the file for a search by an alternate key:

```
READ *, product
POSITION(20, KEY=product, KEYLEN=10, KEYID='PR',
&        MODE='EXACT')
```

Once the file is positioned you can read or write the desired data as in the following statements:

```
READ (20,5) customer, product, quantity
PRINT *, customer, product, quantity
```

Operations on HP-defined Files

Table 5-6 shows the operations that you can perform on HP-defined files. For additional information about $RECEIVE and process files, see Section 14, Interprocess Communication.

Table 5-6. Valid Operations on HP-defined Files

File Type	Read	Write	Rewind	Backspace	Endfile	Position
Magnetic tape	Y	Y	Y	Y	Y	N
Card reader	Y	N	N	N	N	N
Printer, console	N	Y	N	N	N	N
Terminal	Y	Y	N	N	N	N
Process	Y	Y	Y	N	N	N
$RECEIVE	Y	Y	N	N	N	N
EDIT format file	Y	Y*	Y	Y*	Y*	Y*
Unstructured	Y	Y	Y	Y	Y	Y
Entry-sequence	Y	Y	Y	N	N	N
Key-sequence	Y	Y	N	N	N	Y
Relative	Y	Y	Y	Y	N	Y

* Operation is not valid in a FORTRAN program that runs as a NonStop process

Considerations

If your program uses a POSITION statement for a relative file, you cannot rewind or backspace the file after executing the POSITION statement.

Control Specifiers in I/O Statements

I/O statements are described in Section 7, Statements.

Table 5-7 on page 5-25 describes the most commonly used I/O control specifiers.

Table 5-7. I/0 Control Specifiers (page 1 of 2)

Control Specifier	Meaning
UNIT = *unit*	Is an integer expression ranging from 1 through 999 that specifies the FORTRAN unit to use. *unit* can be an asterisk (*) implying the default input unit in a READ statement or the default output unit in a WRITE statement. *unit* can also be the name of a character variable, array, array element, or substring identifying an internal file. If you omit the UNIT= part of this specifier, *unit* must be the first item in the control specifier list.
FMT = *fmt*	Specifies a format to be used for formatted I/O; *fmt* can be: ● A statement label of a FORMAT statement in the same program unit. ● A character expression whose value is the format specification. ● A character array name whose elements, concatenated together, contain the format specification. ● An integer variable that has been assigned the statement number of a FORMAT statement by an ASSIGN statement. ● An asterisk (*) indicating list-directed I/O. If *fmt* is a character array name or a character expression, its value must be a format specification (as described in FORMAT Statement on page 7-39) except that it does not begin with the word FORMAT. It must, however, include the parentheses that enclose the list of format items, as in the following example: `FMT = '(I5, A8)'` If you omit the FMT= part of this specification, *fmt* must be the second item in the control list, and the unit specifier, without the characters UNIT=, must be the first item in the list.
REC = *rn*	Specifies the number of the record to be read or written to a file that is opened for direct access. This is applicable to unstructured, relative, and EDIT format files.

Table 5-7. I/0 Control Specifiers (page 2 of 2)

Control Specifier	Meaning
END = *label*	Specifies the label of an executable statement to which FORTRAN transfers control if an end of file is encountered during an input operation. You can use this specifier only with the READ statement.
ERR = *label*	Specifies the label of an executable statement to which FORTRAN transfers control if an error condition is encountered during I/O processing.
IOSTAT = *ios*	Specifies an integer variable or array element. After the execution of an I/O statement, *ios* returns zero if no error occurred, a file system error, an error defined by the FORTRAN run-time library, or -1 if an end of file was encountered. If *ios* is less than 10000, ios is a file system-defined error number. If *ios* is greater than 10000, *ios* is a FORTRAN run-time library-defined error number and the actual error number is ios - 10000. If an I/O statement does not specify an ERR or an IOSTAT option and an I/O error occurs, FORTRAN terminates the program and displays a diagnostic message.

I/O Lists

The I/O list of an I/O statement specifies the items to transfer and the order of transmission. Separate list items with commas. FORTRAN transmits the items sequentially from left to right.

A list item can be a variable name, an array or array element name, a character substring name, a RECORD, a RECORD field name, or an implied DO list. In a PRINT or WRITE statement, a list item can also be an expression. An array name in an I/O list specifies the entire array in the order in which it is stored.

Considerations

- A standard-conforming program should not use an expression in an I/O list to reference a function if such a reference would cause any I/O operations to be executed or if the reference would cause the value of any element in the I/O list to change.
- You cannot use assumed-size arrays without subscripts in an I/O list.

Using Implied DO Lists

An implied DO list has the following form:

```
( [(]... ar-nam1( var1 [, var2 ]... )
[, ar-nam2( var1 [, var2 ]... ) ]... ,
var1 = iexp1, fexp1°[, incr1 ] )
[, var2 = iexp2, fexp2°[, incr2°] ) ]...
```

ar-name

is the name of an array.

var

is an integer, real, or double precision control variable. It must be a simple variable, not an array element or a RECORD component.

iexp

is an expression that specifies the initial value of each *var*.

fexp

is an expression that specifies the final value of each *var*.

incr

is an expression that specifies the increment value of each var. The default value for *incr* is 1.

You can specify a DO list both in DATA statements and in data transfer statements.

The following example uses an implied DO list to initialize the array INVENTORY to 0:

```
DATA (inventory(i), i = 1, 25) / 25 * 0 /
```

You can use an implied DO list to initialize partial arrays. For example, the following DATA statement initializes the first 10 elements and the last 10 elements of the array PAY(30) to 0:

```
INTEGER pay(30)
DATA (pay(k), k = 1,10), (pay(k), k = 21,30) /20 * 0/
```

You can read the elements of multidimensional arrays using an implied DO list. For example, the following READ statement reads all elements of the two-dimensional array A:

```
INTEGER a(2, 4)
READ(10,100) ((a(i,j), i = 1,2), j = 1,4)
```

You can use a WRITE statement to write the elements of more than one array. The following WRITE statement writes the 100 elements in each of the three arrays A, B, and C:

```
INTEGER a(10, 10), b(10, 10), c(10, 10)
WRITE(9,10) ((a(i,j), b(i,j), c(i,j), i = 1,10), j=1,10)
```

Unformatted I/O

If a data transfer statement does not contain a format specifier in its control list, it is an unformatted I/O statement. Unformatted READ and WRITE statements transfer data as is between memory and an external device. Each statement transfers exactly one record.

Formatted I/O

Formatted I/O is either list-directed or edit-directed:

- If the format specifier of a data transfer statement is an asterisk (*), the statement is a list-directed I/O statement.
- If the control list of a data transfer statement contains a format specifier (other than an asterisk), the statement is an edit-directed I/O statement.
- The first character of an edit-directed output record controls the vertical spacing for an output device (by default, a terminal, a printer, or a process).

Character	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

You can use the SPACECONTROL specifier in the OPEN statement to cause FORTRAN to treat the first character of such a record as a datum. For additional information, see OPEN Statement on page 7-70.

This subsection describes only list-directed I/O. Edit-directed I/O is described in Section 7, Statements.

List-Directed I/O

FORTRAN uses the format of the data it reads to determine the type of input data, and formats output data according to the values contained in the variables it writes.

You must specify at least one of END= or IOSTAT= if you want your program to continue running if an end of file is encountered.

List-Directed Input

The form of the input field must be acceptable for the type of the input list item. Input data consists of a string of values separated by one or more blanks, by a comma, or by a slash (/). If the separator is a comma or a slash, it can be preceded and/or followed by any number of blanks. The end of a record is treated as a blank character. Input values must not contain embedded blanks nor span records except for character values or complex numbers.

Table 5-8. Input Format in List-Directed I/O

Type	Format
Integer	Integer constant format.
Real	Valid FORTRAN format for real or double precision numbers. If you omit the decimal point, FORTRAN assumes that the decimal point is to the right of the mantissa.
Complex	Two real values, separated by a comma, and enclosed in parentheses. The parentheses are not considered to be separators. You can omit the decimal points in each of the components.

Character	A string of characters enclosed by apostrophes. Use two adjacent apostrophes with no intervening blanks to indicate an enclosed apostrophe. You can continue a character constant from the end of one record to the beginning of the next, for as many records as are needed.
	You can read character values only into character arrays, character variables, and substrings. If the string is shorter than the list item, FORTRAN left-justifies and blank fills the string; if the string is longer than the list item, FORTRAN truncates the rightmost characters; for example:
	`CHARACTER*11 title`
	`READ *, title`
	`Input: 'Quarterly Results'`
	`Value: Quarterly R`

Logical	An optional period, followed by T or F, followed by optional characters which do not include separators.

To repeat a value, enter an integer repeat constant followed by an asterisk and the value to repeat. Do not use blanks in a repeat specification.

```
READ *, j, k
Input: 2*50 Value: 50 50
```

If you input a null in place of a constant, the value of the corresponding list entity is not changed. Indicate a null using a comma as the first character in the input string or two commas separated by blanks; for example:

```
READ *, j, k


Input: 2, 10 Value: 2 10
       , 5          2 5
       1, 7         1 7
       , ,          1 7
```

A slash value separator causes FORTRAN to treat the remaining list elements as nulls and to discard the remainder of the current record.

```
READ *, j, k


Input: 10, 10 Value: 10, 10
       /5             10, 10
```

List-Directed Output

List-directed output transfers data from storage locations specified in the I/O list to a unit in the same way as list-directed input except that null values, slashes, repeated constants, and apostrophes to indicate character values are not produced. The formatter suppresses trailing zeros in the mantissa and leading zeros in the exponent.

List-directed output statements always produce a blank for carriage control as the first character of each output record.

Logical values are output as T or F. Complex values are enclosed in parentheses with a comma separating the real from the imaginary part. The following sample program is an example of list-directed output:

```
PROGRAM example
INTEGER k(5)
COMPLEX a, b
REAL c
DATA a, b, c, k/(6, 1), (3, -2), 1.E-3,5,10,20,40,1/
PRINT *, a, b
PRINT *, c
PRINT *, k
END

Output: (6, 1) (3, -2)
         .001
         5 10 20 40 1
```

I/O Performance

You can increase the execution speed of your FORTRAN programs by using techniques provided by the Guardian file system to read and write files faster.

Sequential Block Buffering

If a file is a structured (relative, entry-sequenced, or key-sequenced) disk file, and its OPEN statement specifies:

```
MODE = 'INPUT'
ACCESS = 'SEQUENTIAL'
PROTECT = 'PROTECTED' or PROTECT = 'EXCLUSIVE'
```

FORTRAN uses the “sequential block buffering” feature of the Guardian file system to read the file. Sequential block buffering can make your program run significantly faster by transferring data from the disk process to the FORTRAN program one block at a time rather than one logical record at a time.

Read-Through Locks

If your FORTRAN program is reading a structured file when another process is updating the file and your program tries to read a record while the updating process has that record locked, your program usually waits until the updating process unlocks the record.

However, your program can read a locked record, without waiting for the lock to be released, by calling the Guardian SETMODE procedure to specify use of the “readthrough locks” feature as follows:

```
CALL SETMODE (FILENUM (u), 4, 6, 0)
```

The SETMODE call tells the operating system to allow your program to read through locks on the specified file. Call SETMODE after you have opened the file, but before the first read from the file.

Then if you read a locked record, instead of waiting until the record is unlocked, your FORTRAN program reads the record immediately and the READ statement’s IOSTAT specifier returns error code 9 (locked record read) but the contents of the data record are delivered intact. FORTRAN considers this code a warning rather than a serious error.

The following example shows how your program might read a record:

```
READ (u, IOSTAT = error) data-list
IF (error .EQ. 9) error = 0
IF (error .NE. 0) handle-error
```

Because this technique defeats the purpose of record locks, your program might read inconsistent data if you read the data before the updating process has completed its work. Use this technique only in programs for which speed is more important than accuracy and in which you understand how the records in the file are updated.

6 Introduction to Statements

This section provides introductory information to Section 7, Statements. Topics covered in this section include:

Executable and Nonexecutable Statements

A FORTRAN statement is either executable or nonexecutable.

- An executable statement causes the FORTRAN compiler to generate machine instructions that execute when you run your program.
- A nonexecutable statement provides information that controls how the FORTRAN compiler compiles your program.

Table 6-1 lists the FORTRAN executable and nonexecutable statements.

Table 6-1. FORTRAN Statements (page 1 of 3)

Statement	Executable or Nonexecutable	Type	Described Under
ASSIGN	Executable	Assignment	ASSIGN
BACKSPACE	Executable	I/O	BACKSPACE
BLOCK DATA	Nonexecutable	Program Unit	BLOCK DATA
CALL	Executable	Control	CALL
CHARACTER	Nonexecutable	Specification	CHARACTER
CHECKPOINT	Executable	Control	CHECKPOINT
CLOSE	Executable	I/O	CLOSE
COMMON	Nonexecutable	Specification	COMMON
COMPLEX	Nonexecutable	Specification	Type Declaration
CONTINUE	Executable	Control	CONTINUE
DATA	Nonexecutable	Assignment	DATA
DIMENSION	Nonexecutable	Specification	DIMENSION
DO	Executable	Control	DO
DOUBLE PRECISION	Nonexecutable	Specification	Type Declaration

Table 6-1. FORTRAN Statements (page 2 of 3)

Statement	Executable or Nonexecutable	Type	Described Under
ELSE	Executable	Control	IF (Block)
ELSE IF	Executable	Control	IF (Block)
END	Nonexecutable	Program Unit	END
ENDFILE	Executable	I/O	ENDFILE
END IF	Executable	Control	IF (Block)
END RECORD	Nonexecutable	Specification	RECORD
ENTRY	Nonexecutable	Specification	ENTRY
EQUIVALENCE	Nonexecutable	Specification	EQUIVALENCE
EXTERNAL	Nonexecutable	Specification	EXTERNAL
FILLER	Nonexecutable	Specification	RECORD
FORMAT	Nonexecutable	I/O	FORMAT
FUNCTION	Nonexecutable	Program Unit	FUNCTION
GO TO	Executable	Control	GO TO
IF	Executable	Control	IF (Arithmetic) IF (Logical) IF (Block)
IMPLICIT	Nonexecutable	Specification	IMPLICIT
INQUIRE	Executable	I/O	INQUIRE
INTEGER	Nonexecutable	Specification	Type Declaration
INTRINSIC	Nonexecutable	Specification	INTRINSIC
LOGICAL	Nonexecutable	Specification	Type Declaration
OPEN	Executable	I/O	OPEN
PARAMETER	Nonexecutable	Specification	PARAMETER
PAUSE	Executable	Control	PAUSE
POSITION	Executable	I/O	POSITION
PRINT	Executable	I/O	PRINT
PROGRAM	Nonexecutable	Program Unit	PROGRAM
READ	Executable	I/O	READ
REAL	Nonexecutable	Specification	Type Declaration
RECORD	Nonexecutable	Specification	RECORD
RETURN	Executable	Control	RETURN
REWIND	Executable	I/O	REWIND
SAVE	Nonexecutable	Specification	SAVE
START BACKUP	Executable	Control	START BACKUP
Statement Function	Nonexecutable	Specification	Statement Function

Table 6-1. FORTRAN Statements (page 3 of 3)

Statement	Executable or Nonexecutable	Type	Described Under
STOP	Executable	Control	STOP
SUBROUTINE	Nonexecutable	Program Unit	SUBROUTINE
THEN	Executable	Control	IF (Block)
WRITE	Executable	I/O	WRITE

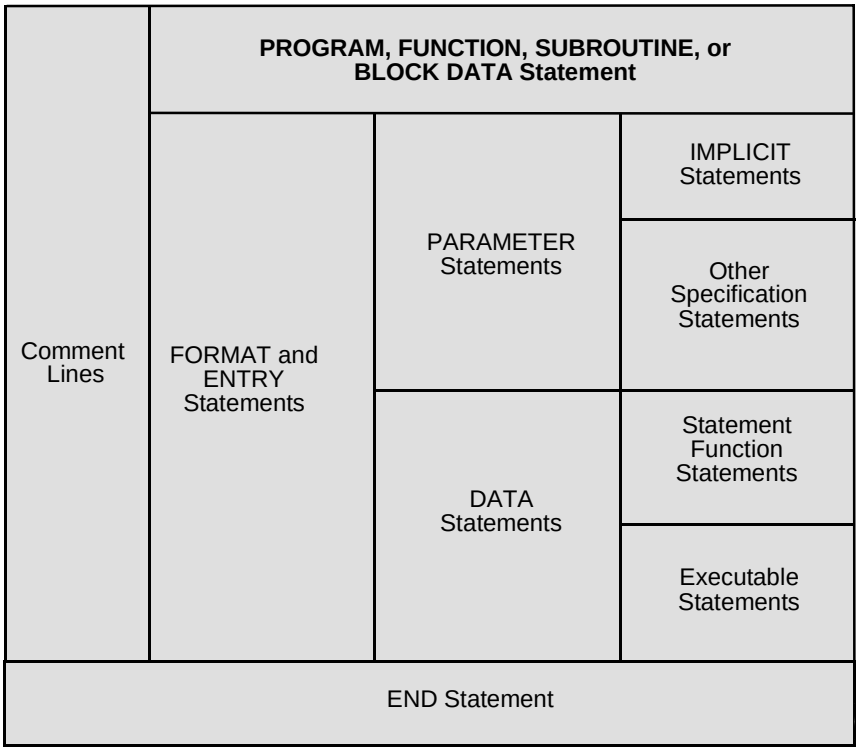
Statement Types

Each FORTRAN statement is classified into one of the following groups:

- Program unit statements
- Specification statements
- Assignment statements Control statements
- I/O statements

Figure 6-1 shows the type of each FORTRAN statement. Vertical items in the table designate varieties of statements that can be interspersed. Horizontal items designate varieties of statements that cannot be interspersed.

Figure 6-1. Order of FORTRAN Statements



VST0601.vsd

Table 6-2 lists the characteristics of each group of statements.

Table 6-2. FORTRAN Statement Types

Type	Action
Program unit	Nonexecutable. Marks the beginning or end of a program unit.
Specification	Nonexecutable. Specifies the characteristics of the user-defined symbolic names used in the program.
Assignment	Executable. Defines or redefines the values of variables in a program.
Control	Executable. Modifies the normal sequential flow of execution in a program.
I/O	Executable. Transfers data between your program and internal or external files, and returns information about the status of files.

Statement Order

The order of statements in a FORTRAN program can determine whether the program compiles successfully. The following list describes the order in which you can specify statements in your FORTRAN program.

- A PROGRAM statement can appear only as the first statement of a main program.
- An END statement is required as the last statement of each program unit.
- The first statement of a subprogram must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.
- Comments can appear anywhere in a program. FORTRAN associates comments that follow an END statement with the program unit, if any, that begins after the END statement.
- Specification statements must precede executable statements, statement function statements, and DATA statements in a program unit.
- Within the specification statements, IMPLICIT statements must precede all other specification statements except PARAMETER statements.
- Statement function statements must precede all executable statements.
- If the default type of the symbolic name of a constant is not the correct type, you must specify the symbolic name's type prior to its first appearance in a PARAMETER statement.
- The PARAMETER statement must precede any statement that refers to the entity defined by the PARAMETER statement.
- FORMAT statements can appear anywhere in a program unit.
- ENTRY statements can appear anywhere in a program unit except in the body of a block-IF or DO statement.

Figure 6-1 on page 6-3 shows the order in which statements can appear in your program.

Statement Labels

A statement label uniquely identifies a statement within a program unit and must not be defined more than once in that unit. A statement label consists of from one to five digits, ranging in value from 1 to 99999. All digits must appear in columns one to five. Blanks are ignored. The following examples show labels:

```
12334 CONTINUE        <-- Valid (label is "12334")
123 4 CONTINUE        <-- Valid (label is "1234")
123 45CONTINUE        <-- Label is "1234"; "5" specifies that
                          this line is a continuation line
```

A label is known only in the program unit containing it. You cannot reference it from a different program unit. You can label any statement, but only the FORMAT statement and executable statements can be referenced by other statements.

Error Numbers

The following I/O statements include an IOSTAT = *ios* option specifier:

```
BACKSPACE      INQUIRE      READ
CLOSE          OPEN         REWIND
ENDFILE        POSITION     WRITE
```

The error numbers returned in the IOSTAT = *ios* option specifier are different for many errors in FORTRAN programs compiled with ENV COMMON in effect than the error numbers returned to programs compiled with ENV OLD in effect.

In programs compiled with ENV OLD in effect, error numbers returned in ios are file system errors except error numbers 256 through 274, which are formatter error numbers.

In programs compiled with ENV COMMON in effect, if *ios* is less than 10000, the error number in *ios* is a Guardian error number. If *ios* is greater than 10000, the error number in *ios* corresponds to an error detected by the FORTRAN run-time library, and the actual error number is the value in ios minus 10000. For example, if *ios* is 48, the error returned is file system error 48, “Security Violation.” If *ios* is 10257, the error number is 257, “Invalid Parameter Value.”

7 Statements

This section describes the FORTRAN language statements. Topics covered in this section include:

For a summary of FORTRAN statements and an explanation of statement order and statement types, see Section 6, Introduction to Statements.

Type Declaration Statements

You use type declaration statements to override or confirm implicit typing and to specify the dimensions of arrays.

The appearance of the symbolic name of a constant, variable, array, external function, RECORD field, or statement function in a type declaration statement specifies the data type of that name throughout the program unit.

You can use a type declaration statement to declare the type of an array as well as to specify the array's dimensions. The following statement dimensions the array ZIP and declares it to be an integer array:

```
INTEGER zip (100)
```

You must specify an array's dimensions in a type declaration statement, a COMMON statement, or a DIMENSION statement. An array's dimensions must be declared only once in a program unit.

Considerations

- You can define the data type of an entity only once in a program unit.
- You cannot use type declaration statements to redefine the data type of an intrinsic function.
- You cannot specify a type for the name of a main program, subroutine, or block data program unit.

For additional information about implicit and explicit data types, see Section 2, Language Elements.

Type Declaration Statements—CHARACTER

The CHARACTER statement defines a variable, array, RECORD field, symbolic constant, function name, or dummy procedure as character type.

```
CHARACTER [* len] name [ dimension ] [* len]
[, name [ dimension ] [* len] ]...
```

len

is an unsigned, nonzero positive constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses that specifies the length, in bytes, of *name*.

name

is the symbolic name of a constant, variable, array, RECORD field, function, or dummy procedure.

dimension

is an array bounds specification in the form:

```
( [ lower:] upper [,[ lower:] upper ]... )
```

where *lower* is an integer expression that specifies the lower bound, and *upper* is an integer expression that specifies the upper bound of an array dimension.

Considerations

- *len* must be in the range 1 through 255. If you omit the length specification, FORTRAN assumes the length is one character.
- If *len* is not an integer constant, you must enclose it in parentheses:

```
PARAMETER (word = 10)
CHARACTER * (word) name, city, state
```

- A length specification following an individual *name* specifies the number of characters in that name:

```
CHARACTER name * 20
```

- A length specification immediately following the word CHARACTER defines the length of each *name* that does not specify a length. If *name* is an array, the length specification defines the length of each array element.
- Using an asterisk for *len*

  When you express the length of *name* as an asterisk enclosed in parentheses (*):

  - If a dummy argument uses the (*) length specification, the dummy argument assumes the length of its associated actual argument on each call to the

subroutine or function. If the actual argument is an array name, *len* equals the length of an array element.

- If an external function uses the (*) length specification in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement within the same subprogram. When a calling program unit references this function, *len* assumes the length declared for that external function name in the calling program unit.
- If a symbolic character constant uses the (*) length specification, the constant assumes the length of its corresponding constant expression in the PARAMETER statement. In the following example the symbolic constant name MONTH has a length of 7:

```
CHARACTER *(*) month
PARAMETER (month = 'january')
```

- The length specification for a character function declared in any program unit that references the function must agree with the length specification in the subprogram that defines the function.

Example

In the following statement, the variables SCHOOL, CITY, and STATE each have a declared length of 8, and the variable NAME has a declared length of 20.

```
CHARACTER * 8 name * 20, school, city, state
```

Type Declaration Statements—LOGICAL

The LOGICAL statement defines a variable, array, symbolic constant, RECORD field, function name, or dummy procedure name as logical type.

```
LOGICAL name [ dimension] [, name [ dimension] ]...
```

name

is the symbolic name of a constant, variable, array, RECORD field, function or dummy procedure.

dimension

is an array bounds specification in the form:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

where *lower* is an integer expression indicating the lower bound, and *upper* is an integer expression indicating the upper bound of an array dimension.

Considerations

See the description of the LOGICAL*4 directive in Section 10, Compiler Directives. For information about two-word logical types and standard conformance, see Section 2, Language Elements.

Example

The following statement declares that the variable CONDITION is of logical type:

```
LOGICAL condition
```

Type Declaration Statements—NUMERIC

NUMERIC type declaration statements specify the numeric type of a symbolic constant, variable, array, RECORD field, function name, or dummy procedure name.

```
{INTEGER          }
{INTEGER*2        }
{INTEGER*4        }
{INTEGER*8        } name [ d ] [, name [ d ] ]...
{REAL             }
{DOUBLE PRECISION }
{COMLEX           }
```

name

is the symbolic name of a constant, variable, array, RECORD field, function, or dummy procedure.

d

is an array bounds specification in the form:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

where *lower* is an integer expression indicating the lower bound, and *upper* is an integer expression indicating the upper bound of an array dimension.

Considerations

You can declare the length of an integer variable explicitly by using an INTEGER*2, INTEGER*4, or INTEGER*8 type declaration statement. If you declare a variable (explicitly or by default) as INTEGER without a length specification, its length is determined by the INTEGER* *n* compiler directive in effect for that program unit.

For additional information about numeric type declarations, see Section 2, Language Elements.

Examples

The following statement declares EXPENSE as a double precision variable:

```
DOUBLE PRECISION expense
```

The following statement declares the variable CURRENT as a complex variable:

```
COMPLEX current
```

The following statement declares the variables POPULATION and CONSUMPTION as doubleword integers.

```
INTEGER*4 population, consumption
```

Statement Function

A statement function is a nonexecutable single-statement computation.

```
function-name ( [ dmy [, dmy ]... ] ) = expression
```

function-name

is an identifier that specifies the function name.

dmy

is a variable that is a statement function dummy argument.

expression

is an arithmetic or character expression.

Considerations

- A statement function has the scope of its containing program unit. It must appear following all declaration statements and before the first executable statement of that program unit.
- The relationship between *function-name* and *expression* must conform to the rules for assignment statements. See the Assignment Statement on page 7-7.
- The actual arguments passed to a statement function must agree in number, order, and type with the dummy arguments of the statement function.
- FORTRAN executes a function statement as follows:
  1. It evaluates actual arguments that are expressions.
  2. It associates actual arguments with their corresponding dummy arguments.

3. It computes a value for expression.
4. It converts the resulting value to the data type of the function name.

When execution of the statement function terminates, FORTRAN assigns the resulting value to the function name in the referencing statement.

In the following example, FORTRAN uses the value of (DIAMETER/2) in the referencing statement to obtain a value for the statement function:

```
volume(radius) = 4.189 * radius**3 <-- statement function
sphere = volume(diameter/2)        <-- referencing
                                       statement
```

- Note that statement function definitions and assignment statements with an array element on the left of the equals sign look exactly the same. The difference between them depends upon whether the name at the beginning of the statement has been declared as an array name. Thus, the compiler interprets the following statement

  ```
  name (a, b) = expression
  ```

  as a statement function definition if name has not been declared as an array name.
- Each symbolic name in the expression of a statement function can reference either a variable within the same program unit or a dummy argument of the statement function.
- You can reference a dummy argument of a function or subroutine statement in the expression of a statement function within the same subprogram.
- You can reference the dummy argument of an ENTRY statement in the expression of a statement function only if the ENTRY statement precedes the statement function in the same subprogram.
- A statement function’s dummy argument names have the scope of the statement function definition.

Example

The following example defines and references the statement function NETPAY to calculate the net pay of each employee:

```
REAL netpay, salary, tax, insure
netpay(salary,tax,insure) = salary-(tax*salary)-insure
DO 5 j = 1,n
   READ (*,*) employee, s, t, x
   salary = netpay(s,t,x)
   WRITE (*,*) employee, salary
5 CONTINUE
```

Assignment Statement

An assignment statement defines the value of an arithmetic, character, or logical entity.

name

is the name of a variable, array element, substring, RECORD, RECORD field, RECORD array element, or RECORD substring.

arithmetic-expression

is an arithmetic expression.

character-expression

is a character expression.

logical-expression

is a logical expression.

Considerations

- Arithmetic assignment statement

  When your program executes an assignment statement, FORTRAN evaluates *arithmetic-expression* according to the rules described in Section 3, Expressions. It then converts *arithmetic-expression* to the type of name, and stores the value of *arithmetic-expression* in name. If name is too small to

contain the value, arithmetic overflow occurs. The type of name need not be the same as the type of *arithmetic-expression*, but both must be arithmetic types. If either name or *arithmetic-expression* is type character, both must be type character. If either name or *arithmetic-expression* is type logical, both must be type logical.

- Character assignment statement

  When your program executes a character assignment statement, FORTRAN evaluates *character-expression* to produce a character string, and stores the string in name. The type of name must be character, but it can have a different length from character-expression.

  If name is shorter than *character-expression*, FORTRAN truncates the excess rightmost characters of *character-expression*. If name is longer than *character-expression*, FORTRAN pads *character-expression* with blanks (on the right) until its length is equal to that of name.

  The character expression cannot refer to any character position included in name. For example, the following statement is invalid:

  ```
  name(3:9) = name(2:7)
  ```

  because the string NAME(3:7) appears on both sides of the equals sign. The following expression is valid because the two substrings are separate entities:

  ```
  name(1:3) = name(4:6)
  ```

  If *name* refers to a substring, only that substring is affected by the assignment statement.

- Note that in the case of character assignments, intermediate results are not stored in temporary locations. The evaluation of a character expression progresses from left to right, character by character to the receiving location. In the following example, the value of A is “abcdeabcde” not “abcdefghij”:

  ```
  CHARACTER*10 A
  A = 'fghij'
  A = 'abcde' // A
  ```

  For additional information about character expressions, see Section 3, Expressions.

- Logical assignment statement

  When a logical assignment statement executes, FORTRAN evaluates *logical-expression* and assigns its value to *name*, which must be of logical type.

Examples

In the following example, the expression BALES * 345.87 is evaluated, converted to a double precision number, and stored in the variable WEIGHT.

```
DOUBLE PRECISION weight
weight = bales * 345.87
```

In the following example, the variable X is typed as a logical variable and assigned a logical value of .TRUE..

```
LOGICAL x
x = .TRUE.
```

The following example stores in FULLNAME the concatenation of the characters stored in FIRST, a blank character, the characters stored in LAST, a comma, a blank, and the first character of MIDDLE.

```
CHARACTER*10 full*25, first, last, middle
fullname = first // ' ' // last // ', ' // middle(1:1)
```

Note that the concatenation produces a 24-character string which is stored in FULLNAME, but, because FULLNAME has a declared length of 25 characters, FORTRAN stores a blank character after MIDDLE(1:1).

ASSIGN Statement

The ASSIGN statement assigns the value of a statement label to an integer variable. You can use the ASSIGN statement to specify the label of an executable statement (see the GO TO Statement on page 7-55), or to specify the label of a FORMAT statement.

```
ASSIGN label TO name
```

label

is the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

name

is the symbolic name of an integer variable. It cannot be an array element.

Considerations

You must define a variable with a statement label value before you reference it in an assigned GO TO statement or as a format identifier in an I/O statement. (See the GO TO Statement on page 7-55.) These two uses are the only valid means of referencing a variable to which you have assigned a statement label.

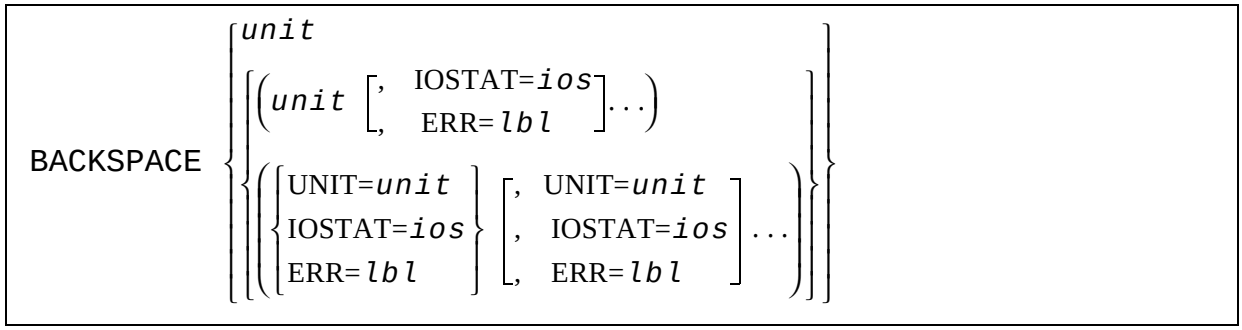
You can use 501 ASSIGN statements within a single program unit.

Example

```
      ASSIGN 10 to J
      GO TO 50
   10 CONTINUE
      .
      ASSIGN 20 TO J
      GO TO 50
   20 CONTINUE
      .
   50 statement
      .
      GO TO J(10,20)
```

BACKSPACE Statement

The BACKSPACE statement backspaces by one record the file connected to a unit. If there is no preceding record in the file, FORTRAN ignores the statement.



unit

is an integer expression from 1 through 999 that identifies an external unit connected for sequential access. The unit must be connected to a magnetic tape, an unstructured file with fixed-length records, an EDIT format file, or a relative file. You cannot backspace entry-sequenced, key-sequenced, or $RECEIVE files.

ios

is an integer variable or integer array element in which FORTRAN returns an error number if an error occurred while executing the BACKSPACE statement. If the BACKSPACE operation is successful, *ios* is zero. For more information about error numbers, see the Error Numbers on page 6-5.

lbl

is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while executing the BACKSPACE statement.

You can write the control specifiers in any order, except that if you omit the UNIT keyword, you must write the unit specifier as the first item in the list.

Considerations

- If the file is an unstructured disk file that does not exist, the BACKSPACE statement creates it.
- Backspacing over records that were written using list-directed formatting is likely to cause an error because in general you do not know how many records are processed by a given list-directed statement.
- After a BACKSPACE statement on an EDIT format file, an INQUIRE statement's NEXTREC option returns the record number of the line preceding the new current record (or -1 if there is none) as the latest record number.
- You cannot backspace an EDIT format file if your program runs as a NonStop process.

  If you specify the ENV COMMON and NONSTOP directives, you cannot backspace an EDIT format file, even if your program is not running as a NonStop process.
- If you backspace unit 5 or unit 6 and you have not already established a connection for the unit, BACKSPACE implicitly opens the unit using default parameters. If you specify ENV COMMON and you backspace unit 5 or unit 6, your FORTRAN routines share access to standard input or standard output, respectively, with routines written in other languages only if the access mode for unit 5 is INPUT and for unit 6 is OUTPUT. However, the default access mode for both units 5 and 6 is I-O. If you want to share access to the file connected to the unit, you must set the unit's access mode to INPUT (unit 5) or OUTPUT (unit 6) before you execute the BACKSPACE statement. You can set the access mode:
  - In a FORTRAN OPEN statement, as in:

    ```
    OPEN( 5, MODE = 'INPUT')
    OPEN( 6, MODE = 'OUTPUT')
    ```
  - In a TACL ASSIGN command, as in:

    ```
    ASSIGN FT005, , INPUT
    ASSIGN FT006, , OUTPUT
    ```

- In a UNIT compiler directive, as in:

```
UNIT (5, INPUT)
UNIT (6, OUTPUT)
```

For more information about using units 5 and 6 as shared files, see the OPEN Statement on page 7-70.

- If a BACKSPACE statement causes unit 5 or unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.
- Error conditions

  If you specify *lbl*, and an error occurs during backspacing, the BACKSPACE statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by lbl. If you also specified *ios*, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *lbl*, and an error occurs during backspacing, your program continues executing with the statement that follows the BACKSPACE statement. You can analyze ios to determine the error that occurred, if any.

  If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
BACKSPACE (3, IOSTAT=iproblem, ERR=450)
BACKSPACE (40)
```

BLOCK DATA Statement

The BLOCK DATA statement marks the beginning of a block data subprogram. A block data subprogram assigns initial values to entities in common blocks.

```
BLOCK DATA [ subprog-name ]
```

subprog-name

is the symbolic name of the block data subprogram.

Considerations

- A block data subprogram is a nonexecutable subprogram. An executable program can include more than one block data subprogram.
- The optional *subprog-name* has the scope of an executable program. *subprog-name* must be different from any global name or local name within the subprogram.

- Observe the following restrictions when using the BLOCK DATA statement:
  - Write the BLOCK DATA statement as the first statement of a block data subprogram.
  - Use only one unnamed block data subprogram in an executable program.
  - Terminate the BLOCK DATA subprogram with an END statement.
  - You cannot initialize RECORD fields in BLOCK DATA subprograms.
  - You can initialize variables and arrays in common blocks only in BLOCK DATA subprograms.
- For additional information about block data subprograms, see Section 4, Program Units.

Example

The following example shows the block data subprogram TAXRATES. It initializes the entities SURCHARGE, VAT, and SALES to the values .05, .25, and .06 respectively.

```
BLOCK DATA taxrates
REAL surcharge, vat, sales
COMMON/taxes/ surcharge, vat, sales
DATA surcharge/.05/, vat/.25/, sales/.06/
END
```

CALL Statement

The CALL statement transfers control to the specified subroutine.

```
CALL subroutine-name [ ( [ arg [, arg ]... ] ) ]
```

subroutine-name

is the name of a subroutine or dummy procedure.

arg

is an actual argument that is an expression, an array name, a RECORD field, an intrinsic function name, an external procedure name, a dummy procedure name, or an alternate return specifier of the form

```
* label
```

where *label* is the label of an executable statement in the same program unit as the CALL statement.

Considerations

- Actual arguments in a CALL statement must agree in number, order, and type with the dummy arguments specified in the SUBROUTINE statement of the called subroutine.
- An actual argument cannot be a character expression involving the concatenation of an operand having a length specification of (*), unless the operand is the symbolic name of a constant.
- You can use a dummy argument of a subprogram as an actual argument in a CALL statement within that subprogram.
- For additional information about the CALL statement and the use of alternate return specifiers, see Section 4, Program Units.
- For information about calling non-FORTRAN routines, see Section 13, Mixed-Language Programming.

Example

In the example below, the CALL statement calls the subroutine CHECKBAL, passing the actual arguments DEPOSIT, WITHDRAWAL, and SERVICE to the subroutine.

```
PROGRAM main
CALL checkbal(deposit, withdrawal, service)
.
END

SUBROUTINE checkbal(d, w, s)
REAL d, w, s
.
END
```

CHECKPOINT Statement

The CHECKPOINT statement establishes a takeover point for a backup process, or transfers the data and environment information needed by the backup process to take over, or both. CHECKPOINT is an HP extension to the ANSI standard.

CHECKPOINT enables you to use the HP fault-tolerant programming facility.

```
CHECKPOINT [([UNIT=]unit [, UNIT=unit]...[, cpt-spec]...)]
           [(cpt-spec [, cpt-spec]...)                   ]
              [ data [, data ]... ]
```

unit

is an integer expression having a value ranging from 1 through 999 that identifies the number of a FORTRAN unit whose status is to be checkpointed. You can specify any number of units in one CHECKPOINT statement.

cpt-spec

is one of the following:

FILENUM = *exp*

exp is an integer expression whose value is the Guardian file number of a file whose status is to be checkpointed. You can specify any number of files in one CHECKPOINT statement.

ERR = *label*

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while executing the CHECKPOINT statement.

BACKUPSTATUS = *status*

status is an integer variable in which FORTRAN returns the status of the backup process after executing the CHECKPOINT statement. Table 7-10 on page 7-102 lists the status codes that can be returned to your program after it executes a CHECKPOINT or START BACKUP statement. Status codes greater than or equal to 1000 indicate that the backup was not successfully started.

STACK = *stack*

stack is a character expression whose value (ignoring any trailing blanks) is either 'YES' or 'NO'. The default value is 'YES'. See Considerations.

`CPLIST = cplist`

is an array that contains a checkpoint list constructed by the Saved Message Utility procedures. You can provide any number of CPLIST specifiers in one CHECKPOINT statement. See Considerations.

data

is a variable name, array name, array element name, RECORD name, or common block name that specifies a data item whose value is to be checkpointed by CHECKPOINT. See Considerations.

Considerations

- If you omit the UNIT keyword from the unit specifier, *unit* must be the first item in the list.
- If the *data* item is the name of a common block, you must enclose it in slashes; for example:

  ```
  CHECKPOINT (10, ERR=100) /accounts/
  ```

- You can identify files to be checkpointed either by their FORTRAN unit numbers or by their Guardian file numbers.
- You must explicitly specify variables, arrays, and RECORDS to be checkpointed in the *data* list:
  - If they are in common blocks
  - If they are local data named in SAVE or DATA statements
  - If you specify STACK = 'NO'

  Local data not named in SAVE or DATA statements is checkpointed automatically if STACK = 'YES' is specified or assumed.
- The data list can include data items with extended addresses. However, the total size of a checkpoint message cannot exceed 32,767 16-bit words.
- STACK specifier.

  If *stack* has the value 'YES' (the default value), FORTRAN checkpoints the memory stack from the initial L register setting to the current S register setting (top of stack), the corresponding portion of the extended memory stack (if any), the entities specified in the data list, and all units specified in UNIT specifiers; this establishes a takeover point for the backup process.

  If *stack* has the value 'NO', FORTRAN checkpoints the entities specified in the data list and the UNIT= specifiers but does not checkpoint the memory stack and does not establish a takeover point. If a failure occurs, the backup uses the takeover point established by the most recent CHECKPOINT statement that specified or defaulted to a 'YES' stack option. This form is useful if you have to

checkpoint large amounts of data to the backup process. For more information on this usage, see Section 16, Fault-Tolerant Programming.

- CPLIST specifier

  The CPLIST specifier enables you to checkpoint changes to saved messages. You must provide a complete checkpoint list in the form of an INTEGER*4 array for *cplist*. The first element is a header for the list and each remaining element is a single entry in the checkpoint list. The following statements declare a CPLIST for programs compiled with ENV OLD in effect:

```
INTEGER*4 cplist1 (0:100)
INTEGER*2 cpinit1 (2)
EQUIVALENCE (cplist1,cpinit1)
DATA cpinit1/ 100, 0 /
```

  Each entry in the CPLIST1 array contains one doubleword. If you compile your program with ENV COMMON in effect, each entry is 2.5 doublewords. You must change the upper bound of the CPLIST1 array and the first entry in the DATA statement to specify enough space for the wider entries used when you specify ENV COMMON. If you generalize the preceding statements to:

```
INTEGER*4 cplist1 (0:n)
INTEGER*2 cpinit1 (2)
EQUIVALENCE (cplist1,cpinit1)
DATA cpinit1/ n, 0 /
```

  the following formula determines the value of N:

```
n = INT(( 2.5 * number_cplist_entries ) + .5 )
```

  The following lines show the code you must enter to have the same number of entries, 100, with ENV COMMON in effect as you do with ENV OLD in effect:

```
INTEGER*4 cplist1 (0:250)
INTEGER*2 cpinit1 (2)
EQUIVALENCE (cplist1,cpinit1)
DATA cpinit1/ 250, 0 /
```

  Note that you must change the upper bound of the CPLIST1 array as well as the value stored in CPINIT1(1) by the DATA statement, which specifies the number of doubleword entries in the array.

  CPLIST1 is the checkpoint list and the *cplist* parameter for the routine. The checkpoint list contains one header entry (CPINIT1) and one hundred saved message checkpoint entries. The required number of INTEGER*4 array elements depends on the number of operations the list must record prior to checkpoint. For additional information, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

- For further information, see Section 16, Fault-Tolerant Programming.

Examples

```
CHECKPOINT
CHECKPOINT (ERR=200, BACKUPSTATUS=ierr) accountno
CHECKPOINT (6, STACK='YES') msgnum, replycode
```

CLOSE Statement

The CLOSE statement disconnects a file from a specified unit and specifies the status of the file after disconnection.

```
CLOSE ( close-spec [, close-spec ]... )
```

close-spec

is one of the following:

[UNIT=] *unit*

unit is an integer expression from 1 through 999 that identifies an external unit connected for sequential access. If you omit the UNIT keyword, you must write this specifier as the first item on the list.

IOSTAT = *ios*

is an integer variable or integer array element in which FORTRAN returns an error number if an error occurred while executing the CLOSE statement. If the CLOSE operation is successful, *ios* is zero. For more information about error numbers, see the Error Numbers on page 6-5.

ERR = *label*

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while executing the CLOSE statement.

STATUS = *statstat*

is a character expression with a value of either 'KEEP' or 'DELETE' and which determines whether the file is to be kept or deleted after disconnection. For additional details, see Considerations on page 7-19.

STACK = *stack*

stack is a character expression whose value is either 'YES' or 'NO'. The default value is 'YES'. For additional details, see Considerations on page 7-19.

Considerations

- Use of the CLOSE statement

  You do not need to include the CLOSE statement in the program unit which opened the file.

  After you have disconnected a unit using the CLOSE statement, you can reconnect the unit within the same program to the same or to a different file.

  A CLOSE statement that refers to a unit that does not exist or that has no file connected to it has no effect.

  When a program terminates normally, FORTRAN automatically disconnects any units that you have not explicitly closed.

- Error conditions

  If you specify *label*, and an error occurs during the CLOSE operation, the CLOSE statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by *label*. If you specified *ios*, you can determine the error that occurred by analyzing ios.

  If you specify *ios*, but not *label*, and an error occurs during the CLOSE operation, your program continues executing with the statement that follows the CLOSE statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *label*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

- STATUS = *stat*

  The file disposition specifier, STATUS = *stat*, determines the status of the file after it is disconnected from the specified unit. If you do not include a status specifier, one of the following occurs:

  - If you opened the file with STATUS = 'SCRATCH', FORTRAN deletes the file.
  - FORTRAN always deletes temporary files when the open count for the file reaches zero.

  For all other types of files, FORTRAN keeps the file. If you specify STATUS = 'DELETE', FORTRAN returns file system error 12, “File in Use,” if your process or another process has the file associated with unit open.

- The STACK specifier

  When a fault-tolerant program is being run, execution of a CLOSE statement automatically checkpoints program environment information.

  If you specify STACK = 'YES' (the default value), FORTRAN checkpoints the memory stack from the initial L register setting to the current S register setting (top of stack), the corresponding portion of the extended memory stack (if any), and the

unit specified in the UNIT specifier; this establishes a takeover point for the backup process.

If you specify STACK = 'NO', FORTRAN does not checkpoint the memory stack and does not establish a takeover point. If failure occurs, the backup uses the takeover point established by a previous OPEN, CLOSE, or CHECKPOINT statement that specified STACK = 'YES' or did not specify a STACK option.

Examples

```
CLOSE(25, ERR=500, STATUS='DELETE', STACK='NO')
CLOSE(IOSTAT=error, ERR=100, UNIT=10, STATUS='KEEP')
```

COMMON Statement

The COMMON statement enables multiple program units to share data.

```
COMMON [ / [ cb ] / ] list [[,] / [ cb ] / list ]...
```

cb

is the name of the block containing the entities in *list*. If you omit *cb*, FORTRAN places the entities in blank common.

list

is a list of entities to be included in the common block. Separate list items with commas. An entity can be a variable name, an array name with or without dimensions, or a RECORD name with or without a dimension.

Considerations

- Blank and named common

  There are two types of common storage: unnamed or “blank” common and named common. Before executing a program, FORTRAN allocates a block of storage for blank common and a block of storage for each named common block.

  If you omit the block name, all the entities in the associated *list* are stored in the blank common block. If the first specification in the COMMON statement is for blank common, you can also omit the slashes.

- Character, numeric, and logical data in common blocks

  You can place numeric and logical type entities in the same common block. A common block containing character data cannot contain any other data type.

- Multiple declarations of the same block

If you declare a common block name more than once in the same program unit, the compiler treats each such common block as a continuation of the first declaration. The following statements:

```
COMMON /tax/ jan,feb,march,//tax rate
COMMON /tax/ april,may,//surcharge
```

are equivalent to the single statement:

```
COMMON /tax/ jan,feb,march,april,may,//tax rate,surcharge
```

- Storage of items in common blocks

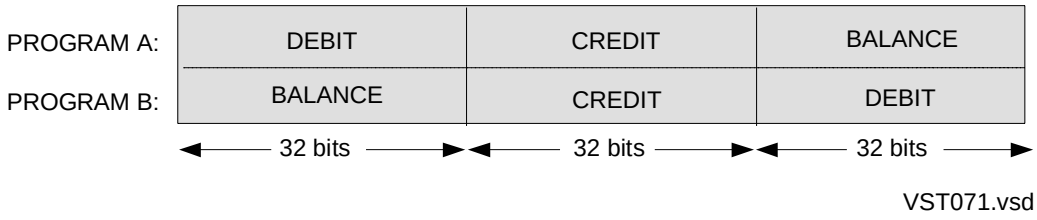
  Each program unit references data items in its common blocks according to the layout it declares. Thus, if program unit A includes the statement

  ```
  COMMON /customer/ debit, credit, balance
  ```

  and program unit B includes the statement

  ```
  COMMON /customer/ balance, credit, debit
  ```

  the data items in the CUSTOMER common block have the following form:



VST071.vsd

  Program A and program B both declare a common block named CUSTOMER that contains three 32-bit REAL variables. The diagram shows that:

  - Program A's variable DEBIT and program B's variable BALANCE share the same 32 bits.
  - Program A's variable CREDIT and program B's variable CREDIT share the same 32 bits.
  - Program A's variable BALANCE and program B's variable DEBIT share the same 32 bits.

  If program unit A modifies the value of DEBIT, it also modifies the value of BALANCE in program unit B. If this is not the effect sought, be careful to maintain consistent naming of common storage items.

- Equivalencing entities in common blocks

  Entities declared in COMMON statements cannot be equivalenced to each other, but a variable, array, or array element that is not in a COMMON block can be equivalenced to a data item that is in a common block. Although an EQUIVALENCE statement must not attempt to expand a common block by adding entities to storage ahead of the first entity in the block, it can extend a common

block beyond its last storage location. The following examples illustrate this restriction:

Illegal	Legal
`COMMON xitem`	`COMMON xitem`
`REAL price(5)`	`REAL price(5)`
`EQUIVALENCE(xitem,price(5))`	`EQUIVALENCE(xitem,price(1))`

You cannot equivalence fields of two different common blocks in the same program unit.

- Common block size

  The size of a common block is equal to the combined lengths of the entities it contains, plus the additional storage space (if any) associated with it by EQUIVALENCE statements.

- Common storage and memory management

  The LARGECOMMON, HIGHCOMMON, and EXTENDCOMMON directives affect the addressing and storage of entities in common blocks. For additional information, see Section 12, Memory Organization.

- Observe the following constraints when using the COMMON statement:
  - You cannot use the same symbolic name in more than one common block within a program unit.
  - You cannot include function names in common blocks.
  - You cannot include names of dummy arguments in common blocks.
  - If you include a RECORD name in a common block, it must be the only item in that block, and the block must be named.
  - A RECORD cannot have more than one dimension.
  - If a common block contains type character data it must not contain data of any other type.
- If a program's source code must be compatible with other FORTRAN compilers, you must also observe the following restrictions:
  - Entities in a named common block can become undefined on execution of a return from a subprogram. This is not the case with blank common.
  - A named common block must be the same size in all program units that declare it. The blank common block can be different sizes in each program unit that declares it.

Example

The following example shows how the main program unit shares data with a subroutine and function subprogram through the use of common blocks. The same area of blank common is used in the main program and in the function COST OF SALES. The common block SALES is used in the main program and in the subroutine SALESREPORT.

```
PROGRAM main
COMMON product, price
COMMON /sales/salesman, commission
.
END

SUBROUTINE salesreport(x, y)
COMMON /sales/salesrep, commission due
.
END

FUNCTION cost of sales ()
COMMON item, price
.
END
```

CONTINUE Statement

The CONTINUE statement does nothing except provide a location for a statement label. It is most often used as the last statement of a DO loop.

```
CONTINUE
```

Considerations

The CONTINUE statement is an executable statement. You can place it anywhere in the executable statement portion of a program without affecting the sequence of execution.

Typically, you use a CONTINUE statement as the last statement of a DO loop, but it is required at the end of a DO loop only if a GO TO or IF statement would otherwise be the last statement of the loop.

Example

In the following example, the PRINT statement executes when J is greater than 10:

```
      sum = 0
      DO 100 j = 1, 10
         sum = sum + 1
  100 CONTINUE
      PRINT *, sum
```

DATA Statement

The DATA statement assigns initial values to variables, arrays, array elements, and substrings at compile time.

```
DATA list / data / [ [,] list / data / ]...
```

list

is a list of entities separated by commas. An entity can be a variable name, array name, array element name, substring name, or an implied DO list.

data

is:

value [, *value*]...

value

is:

{ *datum* | *count***datum* }

datum

is a constant or symbolic constant. You can use Hollerith constants to specify *datum*.

count

is an unsigned, non-zero, integer constant or symbolic name specifying a repeat count for *datum*. For additional information, see Appendix C, Converting Programs to HP FORTRAN.

Considerations

- DATA statements must follow any specification statements that define the entities initialized by the DATA statement. Except for this restriction, DATA statements can appear anywhere in a program unit.
- *list* cannot include the name of a RECORD, dummy argument, function, entry point, or entity in blank common.

  list can include names of entities in a named common block only within a BLOCK DATA subprogram.
- Each *list* must contain the same number of items as the corresponding data. In the following example, A is initialized to 3.0, B is initialized to 4.0, and C is initialized to the string “Total Revenue”:

```
CHARACTER c*20
DATA a, b, c/ 3.0, 4.0, 'Total Revenue'/
```

- Corresponding *list* and *data* items must agree with respect to type. If the length of a *list* character item exceeds the length of its corresponding *data*, the excess characters are initially defined as blanks. If the length of a *list* character is less than its corresponding *data*, the trailing data characters are truncated.
- The DATA specification statement accepts real numbers in decimal format as well as in exponential D and E formats. For example:

```
DOUBLE PRECISION weight
DATA number, weight/3, 1.31D3/
```

- If you specify an unsubscripted array name as a *list* item, the corresponding *data* must contain one value for each element of that array. Values are assigned to the array elements in a predefined progression, with the leftmost subscript varying most rapidly.
- The repeat count form

  If you need to initialize a series of variables to a single value, you can use the repeat count form of the DATA statement. The following statement stores the value of 98.6 in the variables A, B, C, and D:

```
DATA a, b, c, d / 4 * 98.6 /
```

  To initialize each element of the array INVENTORY(100) to 0, enter:

```
DATA inventory / 100 * 0 /
```

- The implied DO list form

  See Using Implied DO Lists on page 5-27.

Examples

```
DATA stock,rate,high,low/3500,15.25,42.75,13/


CHARACTER * 12 headings(5)
DATA headings/'April','May','June','July','August'/
```

DIMENSION Statement

The DIMENSION statement declares an array name and the number and size of its dimensions.

```
DIMENSION name ( dimension) [, name ( dimension) ]...
```

name

is the symbolic name of an array or a RECORD.

dimension

is the array bounds specification in the form:

```
[ lower:] upper [, [ lower:] upper ]...
```

lower

is an integer expression that specifies the lower bound of a dimension. *lower* must be less than or equal to *upper*. *lower* defaults to one if you omit it.

upper

is an integer expression that specifies the upper bound of the dimension.

Considerations

- You can declare an array's dimensions only once in a program unit.
- You can specify an array's dimensions in a COMMON statement or type statement, instead of in a DIMENSION statement.
- The number of [*lower*:] *upper* pairs in an array's dimension specification establishes the number of dimensions of the array.
- An array can have up to seven dimensions.
- A RECORD cannot have more than one dimension.
- The bounds of an array can be positive, negative, or zero.
- For information about assumed-size arrays or adjustable dimensions, see Section 4, Program Units.

- For additional information about array size and storage, see Section 2, Language Elements.

Examples

The following statements declare and dimension a 10-element array, in which each array element contains 15 characters:

```
CHARACTER item*15
DIMENSION item(0:9)
```

The following statement declares a two-dimensional array:

```
DIMENSION numbers(5, 11)
```

DO Statement

The DO statement specifies a DO loop that repeats execution of one or more statements.

```
DO label [,] var = iexp, fexp [, incr ]
```

label

is the label of an executable statement called the terminal statement of the DO loop.

var

is an integer, real, or double precision control variable. It must be a simple variable, not an array element or a RECORD component.

iexp

is an expression that specifies the initial value of *var*.

fexp

is an expression that specifies the maximum value of *var* within the loop.

incr

is an expression that specifies the increment value of var. The default value for *incr* is 1.

Considerations

- A DO loop includes all the executable statements following the DO statement up to and including the terminal statement.
- Execution of the DO loop proceeds as follows:

1. The expressions *iexp*, *fexp*, and *incr* are evaluated and converted to the type of the control variable *var* if necessary.
2. The control variable is assigned the value of *iexp*.
3. The iteration count is calculated according to the following expression:

   ```
   MAX(INT(( fexp - iexp + incr)/ incr),0)
   ```

   The INT function truncates the result of the expression to an integer value; the MAX function selects the larger of that value or zero. For example, the statement

   ```
   DO 100 j = 1,27,3
   ```

   generates an iteration count of 9.
4. If the iteration count is not zero, the DO loop executes. If the iteration count is zero, execution continues with the statement following the terminal statement of the DO loop; the control variable retains its most recent value.
5. The control variable is incremented by the value of *incr*.
6. The iteration count is decremented by 1.
7. Steps 4 through 6 are repeated until the iteration count equals zero.

If the DO loop executes zero times, the control variable is equal to *iexp*. Otherwise, the control variable is equal to its most recent value plus the value of *incr*.

If the DO loop becomes inactive before the iteration count equals zero, the control variable retains its most recent value. A DO loop becomes inactive if you execute a GO TO statement that branches outside of the range of the DO loop.

The control variable retains its most recent value if:

- A RETURN statement is executed within the range of the DO loop.
- A STOP statement in the executable program is executed, or execution is terminated for any other reason.

- The last statement of a DO loop must not be any of the following:
  - A nonexecutable statement
  - An unconditional or assigned GO TO statement
  - An arithmetic or block-IF statement
  - An ELSE, ELSE IF, or END IF statement
  - A DO statement
  - A RETURN statement
  - A STOP statement

- An END statement

- Nested DO loops

  When a DO loop contains another DO loop, the arrangement is called nesting. The range of a DO statement can include other DO statements as long as the range of each inner DO is entirely within the range of the containing DO statements.

  The last statement of an inner DO loop must be either the same as that of its containing DO loop or occur before it.

  Example A shows nested DO loops that share the same terminal statement. Example B shows nested DO loops with different terminal statements:

Example A

```
      DO 5 m = 1,5
       .
          DO 5 n = 1,10
       .
    5 CONTINUE
```

Example B

```
      DO 1 j = 1,10
       .
        DO 2 k = 2,20,2
       .
    2 CONTINUE
       .
    1 CONTINUE
```

Example

The following example uses a nested DO loop to calculate the average purchase amount in one business day at several store locations:

```
      REAL purchase, sum, average
      INTEGER customers
      READ (*,*) m
      DO 50 j = 1, m
         SUM = 0
         READ (*,*) customers
         DO 25 k = 1, customers
            READ (*,*) purchase
            sum = sum + purchase
      25 CONTINUE
            average = sum/customers
            PRINT *, 'Average purchase for store', j, '=',
            average
   50 CONTINUE
```

ELSE Statement

The ELSE statement defines the beginning of a block of statements to execute as an alternative to the block of statements that follows an IF or ELSE IF statement. For more information about the ELSE statement, see the IF Statement—Block on page 7-60.

ELSE IF Statement

The ELSE IF statement defines the end of a block of statements that began with an IF statement or ELSE IF statement, and defines the beginning of a block of statements to execute as an alternative to the block of statements that follows the preceding IF or ELSE IF statement. For more information about the ELSE IF statement, see the IF Statement—Block on page 7-60.

END Statement

The END statement identifies the physical end of a program unit.

```
END
```

Considerations

- If a subprogram reaches the END statement during program execution, the subprogram returns to the program unit that called it.
- If a main program reaches the END statement during program execution, FORTRAN terminates your program.
- The END statement is the only FORTRAN statement that you must write on one line; it cannot have any continuation lines.
- If the source file includes comment lines following the END statement of the last (or only) program unit of a compilation, the compiler prints them but otherwise ignores them.

Example

```
PROGRAM main
PRINT*, 'On vacation!'
END
```

ENDFILE Statement

The ENDFILE statement writes an endfile record as the next record of the file connected to the specified unit.

You can write the ENDFILE specifiers in any order. However, if you omit the UNIT keyword when you specify *unit*, *unit* must be the first item in the list.

```
ENDFILE { unit                                                                    }
        { ([UNIT=] (unit°) [, IOSTAT=(ios°)] [, ERR=(label°)]) }
```

unit

is an integer expression from 1 through 999 that identifies an external unit connected for sequential access. The unit must be connected to either a magnetic tape, an unstructured file with fixed-length records, or an EDIT format file.

ios

is an integer variable or integer array element in which FORTRAN returns an error number if an error occurred while executing the ENDFILE statement. If the ENDFILE operation is successful, *ios* is zero. For more information about error numbers, see the Error Numbers on page 6-5.

lbl

designates the label of an executable statement in the current program unit to which ENDFILE transfers control if an error occurs while executing the ENDFILE statement.

Considerations

- Following execution of an ENDFILE statement, the file is positioned beyond the endfile record. You must use a BACKSPACE, POSITION, or REWIND statement to reposition a file before executing any data transfer I/O statements.
- If the file can also be connected for direct access, only those records that precede the endfile record are considered to have been written, and only those records can be read during subsequent direct-access connections to the file.
- Execution of the ENDFILE statement for a file that is connected, but that does not exist, creates that file.
- Error conditions

  If you specify *lbl*, and an error occurs while writing the end of file, the ENDFILE statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by lbl. If you also specified *ios*, you can determine the error that occurred by analyzing *ios*.

If you specify *ios*, but not *lbl*, and an error occurs while writing the end of file, your program continues executing with the statement that follows the ENDFILE statement. You can analyze *ios* to determine the error that occurred, if any.

If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

- Using an ENDFILE statement with EDIT format files

  An ENDFILE statement deletes all lines in the file following the line last read or written.

  After an ENDFILE statement, an INQUIRE statement's NEXTREC option returns a value of -2 as the last record number for an EDIT format file.

  If you last opened the EDIT format file with ACCESS ='DIRECT', the ENDFILE statement has no effect.

- You cannot write an endfile record to an EDIT format file if your program runs as a NonStop process.

  If you specify the ENV COMMON and NONSTOP directives, you cannot write an endfile record to an EDIT format file, even if your program is not running as a NonStop process.

- If you write an endfile record to unit 6 and you have not already established a connection for the unit, ENDFILE implicitly opens the unit using default parameters. If you specify ENV COMMON and you write an endfile record to unit 6, your FORTRAN routines share access to standard output with routines written in other languages only if the access mode for the unit is OUTPUT. However, the default access mode for unit 6 is I-O. If you want to share access to the file connected to unit 6, you must set the unit's access mode to OUTPUT before you execute the ENDFILE statement. You can set the access mode:

  - In a FORTRAN OPEN statement, as in

    ```
    OPEN( 6, MODE = 'OUTPUT')
    ```

  - In a TACL ASSIGN command, as in

    ```
    ASSIGN FT006, , OUTPUT
    ```

  - In a UNIT compiler directive, as in

    ```
    UNIT (6, OUTPUT)
    ```

  For more information about using unit 6 as a shared file, see the OPEN Statement on page 7-70.

- If an ENDFILE statement causes unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.

Examples

```
ENDFILE 40
ENDFILE (40, IOSTAT=error, ERR=300)
```

END IF Statement

The END IF statement defines the end of a block of statements that began with an IF, ELSE IF, or ELSE statement. For more information about the END IF statement, see the IF Statement—Block on page 7-60.

ENTRY Statement

The ENTRY statement provides an alternate entry point for a subprogram. It also enables you to specify an alternative dummy argument list for the subprogram.

```
ENTRY name [ ( [ dummy [, dummy ]... ] ) ]
```

name

is the symbolic name of the entry point.

dummy

is a dummy argument that can be a variable name, an array name, a RECORD name, a dummy procedure name, or an asterisk (*).

Considerations

- Execution of a subprogram normally begins with the first executable statement following the FUNCTION or SUBROUTINE statement. An ENTRY statement enables you to begin execution of a function or subroutine at a location other than the first executable statement of the subprogram. A subprogram can have multiple entry points.
- An ENTRY statement in a function subprogram identifies an entry point that you can reference as an external function. An ENTRY statement in a subroutine identifies an entry point that you can reference as a subroutine.
- You can place an ENTRY statement anywhere after the SUBROUTINE or FUNCTION statement in the subprogram. You cannot, however, place an ENTRY statement in the body of a block-IF statement or the body of a DO loop.
- If there are no dummy arguments, you can write the ENTRY statement as follows:

  ```
  ENTRY name
  ```

When you invoke a function specified by this form, you must reference it as name(). For example:

```
PROGRAM main
x = surcharge()
END

FUNCTION tax (a, b, c)
ENTRY surcharge
END
```

- Dummy arguments

  The dummy argument list in an ENTRY statement can be different from the dummy argument list in the FUNCTION or SUBROUTINE statement of the subprogram in which it appears, or from the argument lists in other ENTRY statements in the same program. However, the actual arguments in any reference to a subprogram entry point must agree in number, order, and data type with their associated dummy arguments.

  An asterisk (*) dummy argument is allowed only in a SUBROUTINE ENTRY statement where it functions as an alternate return specifier. For additional information, see Section 4, Program Units.

- Observe the following restrictions in using the ENTRY statement:

  - An entry point name has the scope of an executable program. It cannot appear both as an entry point name and as a dummy argument in a FUNCTION, SUBROUTINE, or other ENTRY statement. An entry point name can appear in a type statement.
  - You can reference an entry point name in any program unit of the executable program. The ANSI standard does not allow you to reference an entry point name in the program unit that declares it, but HP FORTRAN removes this restriction because it supports recursive subprograms.
  - If an entry point name in a function subprogram is of character type, every entry point name in the subprogram, and the name of the FUNCTION statement, must be of character type, and must specify the same character length: either a numeric value or an asterisk (*) enclosed in parentheses. If any of those names has a length specification of (*), all such entities must have a length specification of (*); otherwise, they must all have a length specification of the same integer value.
  - If the name of the FUNCTION statement is any type other than character, then the types of the entry point names can be the same as or different from the FUNCTION name and each other in any combination, but none can be type character.

- In a function subprogram, an entry point name must not appear in any statement other than a type declaration statement preceding the ENTRY statement for that name.
- A dummy argument name must not appear in any statement in a subprogram other than a type declaration statement prior to its first appearance as a dummy argument in a SUBROUTINE, FUNCTION, or ENTRY statement.
- A dummy argument must not be used in an executable statement if the subprogram is entered via an entry point for which the dummy argument is not in the argument list of the ENTRY point invoked by the subprogram's caller.

Example

If the function below is called by the statement

```
plus = ADD(amt)
```

FORTRAN ignores the ENTRY statement and executes statements 1, 3, and 4. If the program is called by the statement

```
more = ADD1(num, mice)
```

FORTRAN executes statements 2, 3, and 4.

```
      FUNCTION add(a)
1     b = a**2
      GO TO 3
      ENTRY add1(a,b)
2     c = b
3     add = a + c
4     RETURN
      END
```

Within a function subprogram all variables whose names are also the names of entries are associated with each other and with the variable whose name is also the name of the function subprogram. Therefore, in the preceding example, when ADD becomes defined, ADD1 also becomes defined.

EQUIVALENCE Statement

The EQUIVALENCE statement defines the sharing of storage space by two or more entities in a program unit.

```
EQUIVALENCE ( var-list) [, ( var-list) ]...
```

var-list

is a comma-separated list of variable names array names, array element names, character substring names, or RECORD names.

Considerations

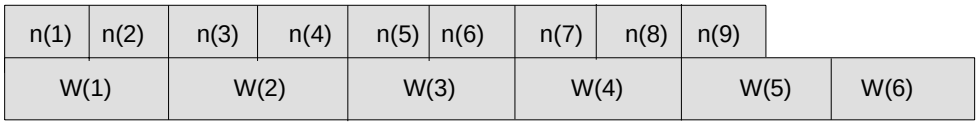
- The compiler assigns the same storage location to all entities in *var-list*. If the equivalenced items are of different data types, the EQUIVALENCE statement does not cause type conversion nor imply mathematical equivalence.
- Equivalencing specifies that all entities in *var-list* share the same first storage word. For character entities, equivalencing specifies that all entities in the list share the same first character storage position.
- If *var-list* includes an array element, the number of subscript expressions must be the same as the number of dimensions declared for that array. An unsubscripted array name in an EQUIVALENCE statement implicitly specifies that equivalencing begin with the first element of that array.
- Subscript expressions, and substring expressions used in *var-list* must be integer constant expressions.
- Observe the following restrictions with the EQUIVALENCE statement:
  - *var-list* cannot contain names of external procedures, dummy arguments, or variable names that are also function names.
  - You cannot use the EQUIVALENCE statement to assign the same storage location to more than one element of the same array.
  - You cannot equivalence an entity of character type to an item of any other data type.
  - You can equivalence RECORDs only to other RECORDs. For additional information, see the Records on page 2-20.

Equivalence With Length Differences

You can equivalence fields of different lengths as long as the EQUIVALENCE statement does not violate the implicit alignment rules of these fields. The following example equivalences the integer array N and the real array W and shows the alignment of the two arrays:

```
DIMENSION n(9), w(6)
EQUIVALENCE (n(1), w(1))
```

DIMENSION n(9), w(6)
EQUIVALENCE (n(1), w(1))



VST0702.vsd

Two elements of the array N occupy the same storage location as one element of the array W.

Equivalencing Items in Common Blocks

Two entities in a common block cannot be equivalenced to each other, but a variable, array, or array element in either a calling program or a subprogram can be made equivalent to an item in a common block. In this case, an EQUIVALENCE statement must not attempt to expand a common block by adding entities to storage ahead of the first entity in the block. You can extend a common block beyond its current last storage location. The following examples illustrate this restriction:

Illegal	**Legal**
`COMMON xitem`	`COMMON xitem`
`REAL price(5)`	`REAL price(5)`
`EQUIVALENCE(xitem,price(5))`	`EQUIVALENCE(xitem,price(1))`

You cannot equivalence elements of two different common blocks.

Example

In the following example, the first five elements of the array RATE share the same storage locations with the five-element array TOTAL:

```
DIMENSION rate(30), total(5)
EQUIVALENCE (rate(1), total(1))
```

EXTERNAL Statement

The EXTERNAL statement declares the name of an external procedure and enables you to use the name as an actual argument.

```
EXTERNAL proc-name [, proc-name ]...
```

proc-name

is the name of an external procedure, dummy procedure, or block data subprogram.

Considerations

If you use an external procedure name or dummy procedure name as an actual argument in a program unit, you must declare it in an EXTERNAL statement in the program unit. When you use *proc-name* in the argument list of a CALL statement or function reference, FORTRAN treats it as a subprogram name rather than a variable or array name.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of an external procedure and you cannot reference the intrinsic function of the same name in that program unit.

Example

In the following example, the function name MULTIPLY must appear in an EXTERNAL statement because it is used as an argument for the external function

```
      CALC:
      PROGRAM main
      EXTERNAL multiply
    4 p = CALC(a, b, multiply)
      END


      FUNCTION CALC(r,s,t)
    1 CALC = s**2 + t(r)
      END
```

FORMAT Statement

The FORMAT statement is used together with formatted I/O statements to write formatted output or read formatted input.

```
FORMAT ( [ format-list ] )
```

format-list

is a list of items, separated by field separators (, /):

$$\left\{\begin{array}{l}[repeat]ed \\ ned \\ [repeat](format-list)\end{array}\right\}$$

repeat

is a nonzero, unsigned, integer constant that specifies the number of successive appearances of ed or format-list.

ed

is a repeatable edit descriptor.

ned

is a nonrepeatable edit descriptor.

Format Control

Formatted data transfer using a format specification initiates format control. Format control depends upon the correspondence between an edit descriptor contained in a format specification and an item in the I/O list of a WRITE, READ, or PRINT statement.

For every I/O list item there must be a repeatable edit descriptor in the format specification, except that for every complex item, there must be two repeatable edit descriptors: one for the real part and one for the imaginary part.

You can use an empty format specification of the form ( ) if you specify no list items. In this case, FORTRAN skips one input record or writes one output record containing no characters.

A format specification is interpreted from left to right. FORTRAN processes a format specification containing a repeat specification as a list containing a repeat number of the format specification. For example, FORTRAN interprets the following specification:

```
FORMAT (2I6, 2F6.2, 2(I5, E3.1))
```

as

```
FORMAT (I6, I6, F6.2, F6.2, (I5, E3.1, I5, E3.1))
```

Blanks are not significant in a format specification unless they are part of a literal string.

Termination of Format Control

For each repeatable edit descriptor in a format specification, format control determines whether there is a corresponding I/O list item. If it finds a corresponding item, it transmits the specified edited information between the item and the record and proceeds to the next item. If it does not find a corresponding item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and you have not specified another item, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and you have not specified another list item, format control terminates.

However, if there is another list item, FORTRAN positions the file at the beginning of the next record and format control reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If format control reverts to a parenthesis that is preceded by a repeat specification, it reuses the repeat specification. In the following examples, the arrows indicate the position of format control in the case of an extra list item:

```
FORMAT  (I5, F4.2, 3(4G9.3, I2))
                            ↑

FORMAT  (I5, F4.2, G13.6)
         ↑
```

After FORTRAN processes each I, F, E, D, G, B, O, Z, L, A, H, or apostrophe edit descriptor, it positions the file after the last character read or written in the current record.

After FORTRAN processes each T, TL, TR, X, or slash edit descriptor, it positions the file as specified by that edit descriptor.

During a read operation, any unprocessed characters of the record are skipped when the next record is read.

Edit Descriptors

Edit descriptors specify data conversions to perform. Repeatable edit descriptors direct the editing of values in a data list. Nonrepeatable edit descriptors provide edit control between the FORMAT statement and one or more records.

Table 7-1 lists the repeatable edit descriptors. Table 7-2 on page 7-42 lists the nonrepeatable edit descriptors. The following subsections provide additional information about the use of these edit descriptors.

Table 7-1. Repeatable Edit Descriptors (page 1 of 2)

Descriptor	Type	Format Example	Sample Output	Description
Ew.d	numeric	E8.1	0.1E+03	Single precision floating-point with exponent
Ew.dEe	numeric	E8.2E2	0.12E+06	Single precision floating-point with explicitly specified exponent length
Fw.d	numeric	F6.0	342.	Single precision floating-point without exponent
Dw.d	numeric	D6.1	.1E+02	Double precision floating-point with or without exponent
Gw.d	numeric	G8.2	1.2	Single precision floating-point with or without exponent
Gw.dEe	numeric	G9.2E2	0.12E+05	Single precision floating-point with explicit exponent length
Iw	numeric	I5	12	Decimal integer
Iw.m	numeric	I5.3	012	Decimal integer with minimum number of digits
Iw.m.b	numeric	I5.3.2	1100	Base b integer with minimum number of digits
Bw	numeric	B3	101	Unsigned binary conversion
Bw.m	numeric	B6.6	000101	Unsigned binary conversion with the minimum number of digits
Ow	numeric	O3	10	Unsigned octal conversion
Ow.m	numeric	O3.3	010	Unsigned octal conversion with minimum number of digits
Zw	numeric	Z2	1D	Unsigned hexadecimal conversion
Zw.m	numeric	Z3.3	21D	Unsigned hexadecimal conversion with minimum number of digits
Lw	logical		T	Logical
A	character	A	blue	Character with data-dependent length
AW	character	A6	yellow	Character with specified length

Table 7-1. Repeatable Edit Descriptors (page 2 of 2)

Descriptor	Type	Format Example	Sample Output	Description
w				Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. Depending on the edit specifier (I, E, F, D, G, B, O, Z or A), the field width can specify the number of positions necessary to accommodate digits, characters, leading blanks, + or - signs, decimal point, and exponent.
d				Unsigned integer constant specifying the number of digits to the right of the decimal point. On output all numbers are rounded.
e				Nonzero unsigned integer constant specifying the number of digits in the exponent.
m				Unsigned integer constant specifying the minimum number of digits to output.
b				Unsigned integer constant specifying the number base for the output data.

Table 7-2. Nonrepeatable Edit Descriptors

Descriptor	Type	Action
SP	Numeric output control	Write plus signs (+).
SS		Suppress plus signs.
S		Suppress plus signs.
BN	Numeric output control	Ignore blanks.
BZ		Treat blanks as zeros.
nX	Tabulation control	Position forward n spaces.
Tn		Transmit next character at nth character position.
TRn		Position forward n spaces.
TLn		Position backward n spaces.
' '	Character	Write the characters within the apostrophes.
nH		Write the n characters that follow the H of the descriptor.
:	Format control	Terminate format control if no I/O list items remain.
/	End of record	Indicate end of current input or output record.
kP	Scale	Establish scaling for numeric editing.

Note: Default values are underlined.

Editing Numeric Data

The edit descriptors:

B	G
D	I
E	O
F	Z

specify the external format of integer, real, double precision, and complex data.

The following general rules apply to numeric editing:

- On input, leading blanks are not significant. The treatment of other blanks is determined either by the BLANK= specifier in an OPEN statement or by any BN or BZ specifiers for that unit. The formatter treats a field of all blanks as zero.
- A decimal point in the input field overrides the decimal point specification of a numeric field descriptor. The following example shows the editing performed for a field described as E5.1:

Input	Stored Value
.5671	0.5671
56.71	56.71
5671	567.1

- On output, positive values are prefixed with a blank unless you use the SP edit descriptor to specify that a plus sign is mandatory. Negative values are prefixed with a minus sign.
- The formatter right-justifies data in the output field for all output conversions. The field width, w, must be large enough to accommodate all characters in the field including, where appropriate, the sign for base and exponent, and decimal point for base. If the number of characters produced by the conversion is less than the specified field width, the formatter inserts leading blanks in the output field unless you specify a minimum number of digits in which case leading zeros are produced as necessary. If the number of characters produced by the conversion exceeds the specified field width, the formatter writes asterisks throughout the field.
- For the B, O, and Z edit descriptors, the data value is treated as unsigned, regardless of its data type. The output data never includes a plus or minus sign.
- Because complex data items must be represented by two floating-point quantities, you must use two conversion elements in the format specification, one for the real part and one for the imaginary part. In the following example the real part of A has

the format specification of F5.2 and the imaginary part has the format specification of E6.3:

```
COMPLEX a
WRITE (6, 5) a, b
5 FORMAT(F5.2,E6.3,F4.2)
```

The I Descriptor

The I *w*, I *w*. *m*, and I *w*. *m*. *b* descriptors specify that the field occupies w character positions. Optionally, they also define the number base for the output data (*b*) and the minimum number of digits (*m*). Base 10 is assumed when b is omitted; 1 is assumed when m is omitted. The corresponding I/O list item can be any numeric type.

On input, the string in the input field must be of an optionally-signed integer constant. An I *w*. *m* descriptor is equivalent to an I w descriptor.

The following example shows an input record (circumflexes designate blanks) and a READ statement that reads the data:

```
135^-12^10^^^5^1
   READ (*,8) i, j, k, l, m
8 FORMAT(BN,I3, I4, I3, I2, I4)
```

After executing the READ statement, the variables contain the following values: I contains 135, J contains -12, K contains 10, L contains 0, and M contains 51.

The output field for the I *w* or I *w*.1. *b* edit descriptor consists of zero or more leading blanks followed by a minus sign (if the value of the datum is negative), or an optional plus sign, followed by the magnitude of the internal value as an unsigned integer constant without leading zeros. An integer constant always consists of at least one digit.

The output field for an I *w*. *m* or I *w*. *m*. *b* descriptor is the same as for an I *w* or I *w*.1. *b* descriptor, except that the unsigned integer constant always consists of at least *m* digits, so the number might contain leading zeros. m must be less than or equal to *w*.

For example, if I contains 33, J contains -99, and K contains 239134, the following statements:

```
   WRITE(*,9) i, j, k
9 FORMAT(I3,2(I6))
```

output the following data (circumflexes indicate blanks):

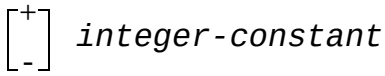
```
^33^^^-99239134
```

The F Descriptor

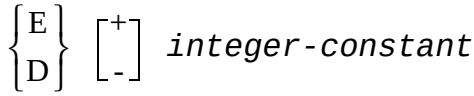
The F descriptor specifies conversion between an internal real or double precision number and an external floating-point number with or without an exponent. The F *w*. *d* form specifies that the field occupies *w* positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign, followed by a string of digits, optionally containing a decimal point. If the input field does not contain a decimal point, FORTRAN interprets the rightmost *d* digits, with leading zeros assumed if necessary, as the fractional part of the value. The basic form can be followed by an exponent in one of the following forms:

- Signed integer constant

- E or D, followed by zero or more blanks, followed by an optionally signed integer constant

  The output field consists of zero or more leading blanks followed by a minus sign if the internal value is negative, or an optional plus sign, followed by a string of digits containing a decimal point that represents the magnitude of the internal value, modified by any scale factor in effect and rounded to *d* fractional digits. A leading zero is inserted to the left of the decimal point only if the value is less than one.

  The following example processes the input record and stores 452.301 in A and 1.E-04 in B. (Circumflexes indicate blanks.)

```
      READ (*,8) a, b
   8 FORMAT ( F7.3, 2X, F5.4)
   Input record: ^452301^^.001^
```

  The following example displays the values shown (circumflexes indicate blanks):

```
      x = .32812
      y = 45.439
      WRITE (*,7) x, y
   7 FORMAT (5X, F4.3, 2X, F4.0)
```

  Output:

```
   ^^^^^.328^^^45.
```

The E and D Descriptors

The E descriptor specifies conversion between an internal real or double precision value and an external floating point number with an exponent. The D descriptor specifies conversion between an internal double precision value and an external floating point number with an exponent.

The E descriptor uses one of the following forms:

E *w*. *d*

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

The field occupies *w* positions, the fractional part of which consists of *d* digits, unless the scale factor in effect is greater than one, and the exponent part consists of *e* digits. The *e* has no effect on input.

The form of the input field for E editing is the same as previously described for F editing.

The D descriptor uses the form D *w*. d and is processed exactly the same as the E *w*. *d* edit descriptor for both input and output.

The form of the output field for E editing with a zero scale factor is:

sign 0 . x1 x2 ... xd *exp*

sign is a minus sign if the value is negative, or an optional plus sign otherwise. x1 x2 ... x*d* are the *d* most significant digits of the value after rounding. *exp* is a decimal exponent in one of the following forms (z is a digit):

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E *w*. *d*	| *exp*| £ 99	E ± z1z2 or ± 0z1z2
E *w*. *d*E*e*	| *exp*| £ (10** *e*) - 1	E ± z1z2 ... z e
D *w*. *d*	| *exp*| £ 99	D ± z1z2 or E ± z1z2 or ± Oz1z2

If you specify a scale factor, *k*, to control decimal normalization:

- If *k* is greater than - *d* and less than or equal to zero, the output field contains exactly \| *k*\| leading zeros and *d* - \| *k*\| significant digits after the decimal point.
- If *k* is greater than zero and less than *d* + 2, the output field contains exactly *k* significant digits to the left of the decimal point and *d* - *k* + 1 significant digits to the right of the decimal point. Other values of *k* are not permitted.

The B Descriptor

The B *w* and B *w*. *m* descriptors convert a data value to a binary (base two) external representation. You can also produce binary conversion using an I *w*. *m*.2 descriptor. The binary digits are 0 and 1. Each binary digit represents one bit of the internal data value.

For example, suppose the data item is type INTEGER*4. The binary conversions produced are shown in Table 7-3 (circumflexes designate blank characters).

Table 7-3. Values Converted With the B Descriptor

Internal Value	Format Descriptor	External Value
0	I10.1.2	^^^^^^^^0
5	I10.1.2	^^^^^^^101
-3	I10.1.2	^^^^^^^-11
0	B10	^^^^^^^^0
5	B10	^^^^^^^101
-3	B10	**********
0	B36	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^0
5	B36	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^101
-3	B36	^^^^11111111111111111111111111111101
0	I36.6.2	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^000000
5	I36.6.2	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^000101
-3	I36.6.2	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-000011
0	B36.6	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^000000
5	B36.6	^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^000101
-3	B36.6	^^^^11111111111111111111111111111101

The O Descriptor

The O *w* and O *w*. *m* descriptors convert a data item to octal (base eight) external representation. You can also produce octal conversion using an I *w*. *m*.8 descriptor. The octal digits are 0 through 7. Each octal digit represents three bits of the internal data value.

For example, suppose the data item is type INTEGER*4. The octal conversions produced are shown in Table 7-4 (circumflexes designate blank characters).

Table 7-4. Values Converted With the O Descriptor (page 1 of 2)

Internal Value	Format Descriptor	External Value
0	I10.1.8	^^^^^^^^0
5	I10.1.8	^^^^^^^^5
-3	I10.1.8	^^^^^^^^-3
0	O10	^^^^^^^^0
5	O10	^^^^^^^^5
-3	O10	**********

Table 7-4. Values Converted With the O Descriptor (page 2 of 2)

Internal Value	Format Descriptor	External Value
0	O20	^^^^^^^^^^^^^^^^0
5	O20	^^^^^^^^^^^^^^^^5
-3	O20	^^^^^^^^37777777775
0	I20.6.8	^^^^^^^^^^^^000000
5	I20.6.8	^^^^^^^^^^^^000005
-3	I20.6.8	^^^^^^^^^^^-000003
0	O20.6	^^^^^^^^^^^^000000
5	O20.6	^^^^^^^^^^^^000005
-3	O20.6	^^^^^^^^37777777775

The Z Descriptor

The Z *w* and Z *w*. *m* descriptors convert a data value to a hexadecimal (base sixteen) external representation. You can also produce hexadecimal conversion using an I *w*. *m*.16 descriptor. The hexadecimal digits are 0 through 9 and uppercase letters A through F. Each hexadecimal digit represents four bits of the internal data value. On input, the lowercase letters a through f are equivalent to the corresponding uppercase letters.

For example, suppose the data item is type INTEGER*4. The hexadecimal conversions produced are shown in Table 7-5 (circumflexes designate blank characters).

Table 7-5. Values Converted With the Z Descriptor (page 1 of 2)

Internal Value	Format Descriptor	External Value
0	I6.1.16	^^^^0
5	I6.1.16	^^^^5
-3	I6.1.16	^^^^-3
0	Z6	^^^^0
5	Z6	^^^^5
-3	Z6	******
0	Z12	^^^^^^^^^0
5	Z12	^^^^^^^^^5
-3	Z12	^^^^FFFFFFFD
0	I12.8.16	^^^^00000000
5	I12.8.16	^^^^00000005
-3	I12.8.16	^^^-00000003

Table 7-5. Values Converted With the Z Descriptor (page 2 of 2)

Internal Value	Format Descriptor	External Value
0	Z12.8	^^^^00000000
5	Z12.8	^^^^00000005
-3	Z12.8	^^^^FFFFFFFD

The G Descriptor

The G *w*. *d* and G *w*. *d*E *e* edit descriptors indicate that the field occupies *w* positions, the fractional part of which consists of d digits, unless a scale factor greater than one is in effect, and the exponent consists of *e* digits.

G input editing is exactly the same as F input editing.

The form of the output field depends on the magnitude of the datum to be represented. If the value is less than 0.1, or equal to or greater than 10** *d*, G editing is exactly the same as E editing. Any scale factor specified controls decimal normalization. If the value is greater than or equal to 0.1 and less than 10** *d*, the scale factor has no effect, and the value determines the editing as shown in Table 7-6.

Table 7-6. Values Edited With the G Descriptor

Not Less Than	But Less Than	Equivalent Editing Effected
0.1	1.0	F(w-n).d, n('b')
1.0	10.0	F(w-n).d-1, n('b')
10.0	100.0	F(w-n).d-2, n('b')
.	.	.
.	.	.
.	.	.
10**(d-2)	10**(d-1)	F(w-n).1, n('b')
10**(d-1)	10**d	F(w-n).0, n('b')

where n is 4 for Gw.d and e+2 for Gw.dEe, and b is a blank.

Table 7-7 shows a comparison of the editing done on output data by similar F and G edit descriptors.

Table 7-7. Comparison of F and G Editing (page 1 of 2)

Value	F13.6 Editing	G13.6 Editing
.01234567	0.012346	0.123457E-01
.12345678	0.123457	0.123457
1.23456789	1.234568	1.23457
12.34567890	12.345679	12.3457

Table 7-7. Comparison of F and G Editing (page 2 of 2)

Value	F13.6 Editing	G13.6 Editing
123.45678900	123.456789	123.457
1234.56789000	1234.567890	1234.57
12345.67890000	12345.678900	12345.7
123456.78900000	123456.789000	123457.
1234567.89000000	**********	0.123457E+07

The P Descriptor

The P descriptor has the form

*k*P

where k is an integer constant called the scale factor. If you omit this specification, FORTRAN assumes a default value of 0 for *k*.

The P descriptor affects the position of the decimal point on input or output. You can use the P descriptor preceding D, E, F, and G format specifications or independently. Once you specify a scale factor, it applies to all subsequent D, E, F, and G descriptors in a FORMAT specification until it is overridden by another scale factor.

On input, for F, E, D, and G editing, the external number is divided by 10** *k* and stored.

On output, for F editing, the external number is the internal number multiplied by 10** *k*. When the number is output, the decimal point stays fixed and the number is adjusted to the left or right depending on whether *k* is positive or negative.

For E and D editing, the kP specification shifts the output coefficient left k places and reduces the exponent by *k*. The scale factor also controls decimal normalization between the coefficient and the exponent such that if *k* is less than or equal to zero, there will be exactly - *k* leading zeros and d+ *k* significant digits after the decimal point.

If *k* is greater than zero, there will be exactly *k* significant digits to the left of the decimal point and d-1+ *k* significant digits to the right of the decimal point.

The scale factor does not affect G editing unless the magnitude of the number output exceeds the range that permits use of the F conversion. In this case the scale factor has the same effect as for E and D editing.

Logical Editing

The L *w* edit descriptor specifies that the external field occupies w positions. An I/O list item matched with an L edit descriptor must be of logical type. On input, the list item will become defined with a logical value; on output, the list item must have been previously defined with a logical value.

The input field consists of optional blanks, optionally followed by a decimal point, followed by an uppercase T for true or an upper-case F for false (lowercase letters are invalid). Additional characters (uppercase or lowercase) can follow the T or F. Note that the logical constants .TRUE. and .FALSE. are valid input forms.

The output field consists of *w*-1 blanks, followed by the letter T or the letter F, depending on the logical value of the internal datum.

Alphanumeric Editing

The A descriptor, apostrophe descriptor, and H descriptor control editing of character data.

The A Descriptor

The A descriptor has the form

```
A or Aw
```

It is used with an I/O list item of type character.

On input, if *w* is greater than the length of the list item, the input quantity is left justified and stored; FORTRAN fills the remaining character positions with blanks. If

w is less than the length of the item, FORTRAN stores the rightmost characters and ignores the rest. If you omit *w*, FORTRAN sets the length of the field equal to the length of the list item.

On output, if *w* is less than the length of the list item, the leftmost characters are output. If *w* is greater than the length of the list item, the characters are output right justified and padded with blanks. For example, the following statements

```
      CHARACTER*8 password
      password = 'sesame'
      WRITE (6,10) password
   10 FORMAT(1x,A10)
```

output the string (circumflexes indicate blanks):

^^sesame^^

Note that one leading blank has been consumed for carriage control.

The Apostrophe Descriptor

The apostrophe edit descriptor has the form of a character constant. It causes the characters (including blanks) within the delimiting apostrophes to be written directly to the output record from the edit descriptor itself. The width of the field is the number of characters contained between the delimiting apostrophes. The formatter writes a single apostrophe to the output device if encounters two adjacent apostrophes within an apostrophe descriptor.

Do not use an apostrophe edit descriptor on input.

The H Descriptor

The *n*H edit descriptor causes the *n* characters (including blanks) that immediately follow the H to be written to the output record directly from the edit descriptor itself, in the same way as the characters in an apostrophe edit descriptor are written to the output device.

You cannot use an H descriptor on input.

Unlike an apostrophe descriptor, adjacent apostrophes in an H descriptor do not have special significance. If you specify two adjacent apostrophes in an H descriptor, they are both written to the output device. However, if within a character constant, you use an H descriptor that contains an apostrophe, you must specify two apostrophes in succession to enter a single apostrophe in the output record. The two successive apostrophes count as a single character with respect to *n* in the *n*H descriptor.

```
WRITE(*, FMT = '(13H Eat at Mom''s)')
```

Positional Editing

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record. On output, these edit descriptors do not, by themselves, cause characters to be written, and therefore do not affect the length of the record. If you write characters to positions at or after the position specified by a positional descriptor, any positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

The T*n* edit descriptor specifies that the *n*th character position of the record will be the next one transmitted. The *n*th character position can be in either direction—forward or backward—from the current position. On input, the T*n* edit descriptor enables your program to process portions of a record more than once, perhaps using different editing each time.

The TL*n* descriptor indicates that the transmission of the next character to or from the record occurs at the character position *n* characters back from the current position. If the current position is less than or equal to position *n*, the transmission of the next character occurs at position one of the current record.

The TR*n* descriptor indicates that the transmission of the next character to or from the record occurs at the character position *n* characters forward from the current position.

The position specified by an X descriptor is forward from the current position. On input, you can specify a position beyond the last character of the record as long as no characters are transmitted from such a position.

Slash Editing

The slash edit descriptor (/) indicates the end of data transfer on the current record.

On input from a file connected for sequential access, a slash descriptor causes FORTRAN to skip the remaining part of the current record and to position the file at the beginning of the next record, which becomes the current record. On output to a file connected for sequential access, the slash descriptor causes FORTRAN to begin a new record, which becomes the last and current record of the file.

For a file connected for direct access, the slash descriptor causes FORTRAN to increase the record number by one, and to position the file at the beginning of the record that has that record number.

Sign Control

The S, SP, and SS edit descriptors control the optional plus sign in the numeric output field.

The SP edit descriptor directs the system to prefix each subsequent output value with a plus sign if the value is greater than or equal to zero. However, an I *w*. *m* or I *w*. *m*. *b* descriptor with *m* equal to 0 produces an all-blank field when the internal value is zero, regardless of the sign control in effect.

The SS edit descriptor suppresses plus signs for positive output values.

The S descriptor is treated as an SS descriptor.

In the following example, the system produces plus signs for the first two integer entities, and suppresses the plus sign for the subsequent real item:

```
FORMAT('1', SP, 2(I5), SS, F5.2)
```

Blank Control

The BN and BZ edit descriptors specify the interpretation of blanks in numeric input fields. These edit descriptors override the OPEN statement's BLANK= specifier.

The BN edit descriptor directs the formatter to ignore embedded blanks in numeric input fields. A field of all blanks is interpreted as having a value of zero.

The BZ edit descriptor specifies that all embedded blank characters in succeeding numeric fields be treated as zeroes.

For example, given the following input record (circumflexes indicate blanks),

```
10^^2^^3
```

the statements:

```
      READ (6, 9) i, y
9 FORMAT (BZ, I3, F5.2)
      PRINT *, i, y
```

output the value of 100 for I, and 20.03 for Y.

Given the preceding input record, the following statements output the value of 10 for J, and 0.23 for B.

```
      READ (6, 11) j, b
11 FORMAT (BN, I3, F5.2)
      PRINT *, j, b
```

FUNCTION Statement

The FUNCTION statement designates the beginning of a function subprogram.

```
[ type ] FUNCTION func-name ( [ dmy [, dmy ]... ] )
```

type

is INTEGER, INTEGER*2, INTEGER*4, INTEGER*8, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [* len].

func-name

is the name of the function subprogram designated by the FUNCTION statement.

dmy

is a variable, array, RECORD, or dummy procedure name that designates a dummy argument.

Considerations

- The FUNCTION subprogram can include multiple entry points (see the ENTRY Statement on page 7-33) and zero or more RETURN statements (see the RETURN Statement on page 7-95). Terminate the FUNCTION subprogram with an END statement.
- You can specify the type of the function name implicitly according to the default FORTRAN convention, or explicitly by a *type* entry preceding the word FUNCTION or in a type declaration statement after the FUNCTION statement. However, if you override the default, you must include that specification in both the calling program and the FUNCTION statement.
- *func-name* has the scope of an executable program and must be different from any other external name.
- You must assign a value to the function name (or any of its other names defined by ENTRY statements) at least once during the execution of the function subprogram.
- *dmy* is unique to the program unit which begins with the FUNCTION statement and ends with the END statement; it must not appear in an EQUIVALENCE,

PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. FORTRAN replaces *dmy* with an actual argument when it executes the function.

- If *dmy* is an array name, you must dimension the array within the function subprogram using a DIMENSION or type statement. For the use of adjustable dimension declarators and for further information about function subprograms, see Section 4, Program Units.

Example

The following example shows the external function OVERTIME, which calculates overtime pay, given the number of hours worked and the hourly rate.

```
FUNCTION overtime (rate, hours)
over = hours - 40
xpay = rate * 1.5
overtime = over * xpay
END
```

GO TO Statement

The GO TO statement transfers control to another statement in the same program unit. FORTRAN provides the following three forms of the GO TO statement:

Unconditional GO TO statement:

```
GO TO label
```

Computed GO TO statement:

```
GO TO ( label [, label ]... ) [,] exp
```

Assigned GO TO statement:

```
GO TO ivar [ [,] ( label [, label ]... ) ]
```

label

is an integer that designates the label of an executable statement within the same program unit.

exp

is an integer arithmetic expression.

ivar

is an integer variable previously assigned a label value in an ASSIGN statement.

Considerations

A GO TO statement must not transfer control to a statement within a DO loop.

Unconditional GO TO

An unconditional GO TO statement transfers program control directly to the statement whose label is specified by the GO TO statement. In the following example, control passes from the GO TO statement to the PRINT statement labeled 300. Intervening statements, if any, are skipped:

```
      GO TO 300
      .
300 PRINT *, answer
```

Computed GO TO

A computed GO TO statement consists of the word GO TO followed by a comma separated list of labels enclosed in parentheses, followed by an arithmetic expression.

Each label must be associated with an executable statement. FORTRAN evaluates the expression and transfers control to the statement whose label's position in the list of labels is equal to the value of the expression. The index of the first label in the list is 1.

In the following example, FORTRAN evaluates the expression and truncates it, if necessary, to an integer result:

```
GO TO (100, 200, 200, 300), k + 1
```

FORTRAN transfers control to statement 100 if k+1 equals 1, to statement 200 if k+1 equals 2 or k + 1 equals 3, and to statement 300 if k+1 equals 4.

If the value of the expression is less than one or greater than the number of labels in the list, FORTRAN transfers control to the statement that follows the computed GO TO statement.

Assigned GO TO

An assigned GO TO statement transfers control to the statement whose label was assigned to *ivar* in an ASSIGN statement that was executed previously in the same run of the program unit containing the assigned GO TO.

Before your program executes an assigned GO TO statement, you must assign a label to *ivar* using an ASSIGN statement. The following example shows an ASSIGN

statement and a GO TO statement that uses the label stored by the ASSIGN statement:

```
      ASSIGN 10 to j
   .
      IF (x .GT. 0) THEN
         ASSIGN 20 to j
      END IF

      GO TO j,(10,20)
      .
   10 CONTINUE
      .
   20 CONTINUE
```

The program branches to 20 if X is greater than 0 and to 10 if X is less than or equal to 0.

Considerations

FORTRAN ensures that all labels referenced in the assigned GO TO statement are in the current program unit and are associated with executable statements. At run time, however, FORTRAN does not check that the label to which your program branches is in the list of labels. FORTRAN transfers control to the statement whose label was specified in the last ASSIGN statement executed before the GO TO statement, regardless of whether that label is in the list of labels for the current assigned GO TO statement.

Examples

The following example shows an unconditional GO TO:

```
   IF (x .GT. y) GO TO 30
   .
   30 difference = x - y
```

The following example shows a computed GO TO:

```
      INTEGER region
   5 READ (*,*) region
      READ (*,*) time
      GO TO (10, 20, 30) region
   10 charge = time * .25
      GO TO 40
   20 charge = time * .27
      GO TO 40
   30 charge = time * .36
      GO TO 40
   40 CONTINUE
```

The following example shows an assigned GO TO:

```
   10 ASSIGN 20 TO k
   20 I = 50
      .
      .(main processing loop)
      .
   250 IF (type .EQ. 'end') ASSIGN 350 TO k
      .
   340 GO TO k, (20,350)
   350 TOT = i + m
      .
      .
```

IF Statement—Arithmetic

The arithmetic IF statement conditionally transfers control to one of three statement labels.

```
IF ( exp) label1, label2, label3
```

exp

is an arithmetic expression.

label1, label2, label3

are integers designating the labels of executable statements in the same program unit. The same statement label can appear more than once in the same arithmetic IF statement.

Considerations

The arithmetic IF statement transfers control to

- *label1* if the value of exp is negative.
- *label2* if the value of exp is zero.
- *label3* if the value of exp is greater than zero.

Example

The following example transfers control to statement 10 if BALANCE is negative, to statement 20 if BALANCE equals zero, and to statement 30 if BALANCE is greater than zero:

```
      READ (8, 99) balance
      IF (balance) 10, 20, 30
10    PRINT *, 'You''re overdrawn by', balance
      GO TO 40
20    PRINT *, 'You have a zero balance'
      GO TO 40
30    PRINT *, 'You have a balance of', balance
40    CONTINUE
```

IF Statement—Logical

The logical IF statement conditionally executes a specified statement.

```
IF ( exp) statement
```

exp

is a logical expression.

statement

is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

Considerations

The logical IF statement executes *statement* if *exp* is true, or continues to the next executable statement if *exp* is false.

Example

```
IF (balance .GT. 10000) account = preferred account
```

IF Statement—Block

The block IF statement is used with the END IF statement and, optionally, with the ELSE IF and ELSE statements, to select groups of statements to execute.

```
IF ( exp) THEN if-block
   [ ELSE IF ( exp ) THEN if-block ]...
   [ ELSE if-block ]
END IF
```

exp

is a logical expression.

if-block

consists of all the executable statements following the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement at the same level as the block IF statement.

Considerations

- A block IF statement begins with an IF statement and ends with an END IF statement. A block IF statement can include ELSE and ELSE IF statements that define the execution of subgroups of statements within the larger block. The following example illustrates the simplest form of the block IF statement:

  ```
  READ (*,*) answer
  IF (answer .EQ. 1) THEN
     deductions = 1000
  END IF
  ```

  If ANSWER equals 1, the program executes the statement between the IF and the END IF statements. Otherwise, control passes to the first executable statement after the END IF statement.

- You can use an ELSE statement to specify a group of statements to execute if the initial condition is false:

```
IF (loan type .EQ. 'consumer') THEN
   rate = normal rate
   premium = 1000
ELSE
   rate = normal rate * .9
   premium = 500
END IF
```

- If more than two conditions must be considered, you can use ELSE IF statements, each followed by an *if-block*:

```
IF(loan type .EQ. 'A') THEN
   rate = normal rate
   premium = 1000
ELSE IF (loan type .EQ. 'B') THEN
   rate = normal rate * .9
   premium = 500
ELSE IF (loan type .EQ. 'C') THEN
   rate = normal rate * .85
   premium = 100
END IF
```

- You can ensure that your program executes at least one *if-block* by using ELSE IF and ELSE statements:

```
IF(loan type .EQ. 'A') THEN
   rate = normal rate
   premium = 1000
ELSE IF (loan type .EQ. 'B') THEN
   rate = normal rate * .9
   premium = 500
ELSE IF (loan type .EQ. 'C') THEN
   rate = normal rate * .85
   premium = 100
ELSE
   rate = normal rate * .75
   premium = 50
END IF
```

- You can code an empty *if-block* if you want to test for a specific value but not take action for that value. In the following example, if LOAN TYPE equals “A”, no statements are executed and control passes immediately to the statement following the END IF statement:

```
IF(loan type .EQ. 'A') THEN
ELSE
   rate = normal rate * .75
   premium = 50
END IF
```

- Do not transfer control into an IF block from outside the IF block.
- You can nest block IF statements. Terminate each block IF statement with an END IF statement. The first IF statement is paired with the last END IF statement, the following IF statement is paired with the next to last END IF statement, and so forth. The nesting scheme is similar to the nested DO loop structure. The following example shows a nested IF block structure.

Example

```
IF (sales .GT. 5000) THEN
   IF (travel .LT. 1000) THEN
      bonus = 100
   ELSE
      bonus = 50
   END IF
ELSE
   bonus = 50
END IF
```

IMPLICIT Statement

The IMPLICIT statement redefines or confirms the default typing of variables, arrays, and functions, based on the first letter of the item's name.

```
IMPLICIT type ( char-list) [, type ( char-list) ]...
```

type

is one of the following data type specifiers: INTEGER, INTEGER*2, INTEGER*4, INTEGER*8, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [* length].

char-list

indicates one or more characters either in the form of a list or a range. Separate list items with commas. Indicate a range as:

first-char - last-char

Considerations

- An IMPLICIT statement declares that the data type of any variable, array, function, or RECORD field whose name begins with the letters in *char-list* is the data type you specify for that letter in an IMPLICIT statement unless you explicitly declare the data type of the symbolic name.
- IMPLICIT statements must precede all declaration statements except PARAMETER statements.
- If you do not specify a length for character-type data, the length is assumed to be one. You must specify the length as an integer constant or an integer constant expression enclosed in parentheses. The specified length applies to all entities in *char-list*. The following statement declares that all symbolic names beginning

with A, X, Y, and Z are character type and represent character data that is 15 characters long.

```
IMPLICIT CHARACTER*15 (a, x-z)
```

- The data type of an item declared in a type declaration statement overrides the type specified in an IMPLICIT statement. The following statements declare that symbolic names that begin with the letters M through Z are type real and names that begin with C are type character with a length of 15:

```
IMPLICIT REAL (m-z), CHARACTER * 15 (c)
INTEGER margin, point of sale
CHARACTER *20 city
DIMENSION net(10)
```

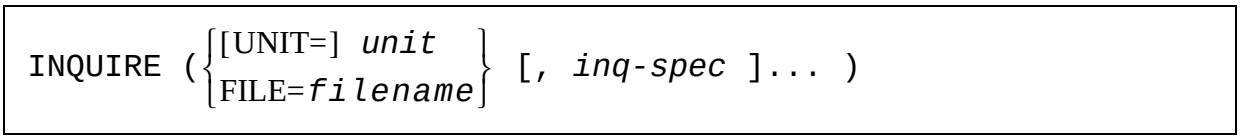
The variable names MARGIN and POINT OF SALE are explicitly declared to be integer, and CITY is a character variable with length 20. The data type of NET defaults to real.

Examples

```
IMPLICIT INTEGER*4(a-e), DOUBLE PRECISION(f-h)
IMPLICIT CHARACTER*20 (i,j,l-o)
```

INQUIRE Statement

The INQUIRE statement ascertains the properties of a file or the properties of the connection of a specified unit.



`[UNIT=]` *`unit`*

is a unit specifier. unit is an integer expression whose value is in the range 1 through 999 that identifies the unit for which information is to be returned.

`FILE =` *`file-name`*

is a file specifier where *`file-name`* is a character expression that specifies the name of the file for which information is to be returned. The file might neither exist nor be connected to a unit. For information about the format of file names, see the *Guardian Programmer's Guide*.

inq-spec

is a keyword followed by an equal sign followed by a variable or array element in which FORTRAN returns the information specified by the keyword. You can specify *inq-specs* in any order. *inq-spec* is one of the following:

IOSTAT = *ios*

ios is an integer variable or integer array element in which FORTRAN returns an error number if an error occurs while executing the INQUIRE statement. If the INQUIRE statement is successful, *ios* is zero. For more information about error numbers, see the Error Numbers on page 6-5.

ERR = *label*

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while executing the INQUIRE statement.

EXIST = *ext*

ext is a logical variable or array element. For an inquiry of a Guardian file, if the specified file exists, ext returns a value of .TRUE.. If no such file exists, ext returns a value of .FALSE.. For an inquiry of a unit, if the specified unit exists, *ext* returns a value of .TRUE.. If the unit does not exist, *ext* returns .FALSE..

OPENED = *open*

open is a logical variable or array element. For an inquiry of a Guardian file, if the specified file is connected to a unit, *open* returns a value of .TRUE.; if the file is not connected, *open* returns a value of .FALSE.. For an inquiry by unit, if the specified unit is connected to a file, *open* returns a value of .TRUE.; if the *unit* is not connected to a file, open returns a value of .FALSE..

NUMBER = *n*

n is an integer variable or array element. If *file-name* or the file referenced by *unit*, whichever you specify, is open, FORTRAN returns the smallest unit number of all open units to the file. If the referenced file is not open, FORTRAN returns the smallest unit number of all units connected to the file.

NAMED = *named*

named is a logical variable or array element. *named* returns a value of .TRUE. if the file has a user-specified name. If the file has a system-assigned, temporary name, *named* returns a value of .FALSE..

NAME = *fname*

fname is a character variable or array element. *fname* is the file's user-specified or temporary name. If the specified unit is not connected, *fname* returns all blanks. See Considerations on page 7-67

ACCESS = *acc*

acc is a character variable or array element. If the file is connected for sequential access, *acc* returns the value 'SEQUENTIAL'; if the file is connected for direct access, *acc* returns the value 'DIRECT'.

SEQUENTIAL = *seq*

seq is a character variable or array element. If sequential access is one of the access methods for the file (regardless of its present connection), *seq* returns the value 'YES'; if the file can never be connected for sequential access, *seq* returns the value 'NO'. If the system cannot determine whether sequential access is permitted, *seq* returns the value 'UNKNOWN'.

HP FORTRAN always returns 'YES' in *seq*.

DIRECT = *dir*

dir is a character variable or array element. If direct access is one of the access methods for the file (regardless of its present connection), *dir* returns the value 'YES'; if the file can never be connected for direct access; *dir* returns the value 'NO'. If the system cannot determine whether direct access is permitted, dir returns the value 'UNKNOWN'.

HP FORTRAN returns:

- 'YES' for unstructured files, EDIT format files, and relative files
- 'NO' for all other files with Guardian device codes less than 12
- 'UNKNOWN' for all other files

FORM = *fm*

fm is a character variable or array element. If the file is connected for formatted I/O, *fm* returns the value 'FORMATTED'; if the file is connected for unformatted I/O, *fm* returns the value 'UNFORMATTED'.

FORMATTED = *fmt*

fmt is a character variable or array element. If the file can be connected for formatted I/O (regardless of its present connection), *fmt* returns the value 'YES'. If the file can never be connected for formatted I/O, *fmt* returns the value 'NO'. If the system cannot determine whether formatted I/O is permitted, *fmt* returns the value 'UNKNOWN'. If there is no connection, *fmt* remains unchanged.

HP FORTRAN always returns 'YES' in fmt.

UNFORMATTED = *unf*

unf is a character variable or array element. If the file can be connected for unformatted I/O (regardless of its present connection), *unf* returns the value

'YES'. If the file can never be connected for unformatted I/O, *unf* returns the value 'NO'. If the system cannot determine whether unformatted I/O is permitted, *unf* returns the value 'UNKNOWN'.

HP FORTRAN always returns 'YES' in *fmt*.

`RECL = rcl`

rcl is an integer variable or array element that returns the record length, specified in bytes, for the file. *rcl* returns a value of zero if the file is not open.

`NEXTREC = nr`

nr is an INTEGER*4 variable or array element. For an open unstructured or relative disk file, the system obtains the record number of the last record read or written, adds 1, and stores the result in *nr*. If no records have been transferred since the file was connected, *nr* returns the value 1.

For an open EDIT format file, nr returns the EDIT line number, multiplied by 1000, of the most recently read or written record. If the last line had the number 12.34, *nr* returns the value 12340. If the file is rewound, *nr* returns -1.

If the most recent record read or written is an end of file, *nr* returns -2. For an open file that is not capable of direct access, *nr* returns -1. For a file that does not exist or is not connected, *nr* returns 0.

`BLANK = blank`

blank is a character variable or array element. If the file is connected for formatted I/O, and was opened with zero blank control or numeric input fields (null is assumed if the OPEN statement contained no blank specifier or there was no OPEN statement for the file), *blank* returns the value 'ZERO'. Otherwise, *blank* returns 'NULL'.

Considerations

- There are two forms of the INQUIRE statement: inquiry by file and inquiry by unit, depending on whether you specify *file-name* or *unit* in the statement. A single INQUIRE statement can execute only one type of inquiry.
- The INQUIRE statement must specify either a file name or a unit, but not both, and not more than one each of the other control specifiers listed in the syntax diagram. The control specifiers can appear in any order, except that if you omit the UNIT= keyword, the unit specifier must be the first item in the list.
- In the INQUIRE statement, FORTRAN returns all string values in upper case letters.
- NAME= *fname*

  The file name assigned to *fname* when you execute an INQUIRE statement is the fully qualified name of the file and is, therefore, not necessarily the same as the

name you specify in the FILE = *file-name* specifier for an inquiry by file. For example, after execution of the following:

```
file id = 'myfile'
INQUIRE (FILE=file id, NAME = title)
```

TITLE contains a fully qualified file name such as $MYVOL.MYSVOL.MYFILE (if the specified file exists), while the value of FILE ID is still MYFILE (either the full name or the short form is acceptable when specifying the file name in an OPEN or INQUIRE statement). If the file you specify is a temporary file, the name returned is the temporary file name (for example, $VOLUME.#1234). To hold the full file name without truncation, *fname* should have a declared length of at least 35 characters.

- Error conditions

  If an error occurs while FORTRAN processes the INQUIRE statement, all the inquiry specifier entities remain unchanged except *ios*.

  If you specify *label*, and an error occurs during the INQUIRE statement, FORTRAN transfers control to the statement identified by *label*. If you also specified ios, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *label*, and an error occurs during the INQUIRE statement, your program continues executing with the statement that follows the INQUIRE statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *label*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
LOGICAL nee, up, am
CHARACTER whosit*34
INQUIRE(UNIT=3,EXIST=am,OPENED=up,NAMED=nee,NAME=whosit)
LOGICAL ready
INTEGER number, howlong
INQUIRE(FILE=infile,OPENED=ready,NUMBER=number,RECL=howlong)
```

INTRINSIC Statement

The INTRINSIC statement identifies the names of intrinsic functions, and enables you to specify intrinsic function names as actual arguments to subprograms.

```
INTRINSIC function [, function ]...
```

function

is an intrinsic function name.

Considerations

- If you use an intrinsic function name as an actual argument in a CALL statement or function reference, you must declare it in an INTRINSIC statement in that program unit.
- Do not use the following intrinsic functions as actual arguments:
  - Type conversion functions (CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, SNGL)
  - Largest/smallest value functions (MAX, MAX1, AMAX0, AMAX1, DMAX1, MIN, MIN0, MIN1, AMIN0, AMIN1, DMIN1)
- Generic function names do not lose their generic property when they appear in INTRINSIC statements.
- Do not include an intrinsic name in more than one INTRINSIC statement in a program unit.
- Do not declare the same name in an INTRINSIC statement and in an EXTERNAL statement in the same program.

Example

```
SUBROUTINE A
INTRINSIC SQRT
 .
CALL number(x, y, SQRT)
 .
END

SUBROUTINE number(a, b, c)
CALL C(A)
CALL C(B)
END
```

OPEN Statement

The OPEN statement associates an existing file with a unit number, or creates a new file and associates it with a unit number.

```
OPEN ( [UNIT=] unit [, open-spec ]... )
```

unit

is an integer expression ranging from 1 through 999. Once defined, the properties of this unit are the same for all program units of the executable program.

open-spec

is one of the following specifiers. You can specify open-specs in any order.

IOSTAT = *ios*

ios is an integer variable or integer array element in which FORTRAN returns an error number if an error occurred while opening the unit. If the OPEN is successful, *ios* is zero. For more information about error numbers, see the Error Numbers on page 6-5.

ERR = *label*

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while opening the unit.

`FILE = fn`

fn is a character expression that specifies the name of the file to connect to unit. It can also be a DEFINE name. For more information about the format of file names, see the *Guardian Programmer's Guide*.

`STATUS = stat`

stat is a character expression with the value of 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. The default is 'UNKNOWN'.

`ACCESS = acc`

acc is a character expression with the value of 'DIRECT' or 'SEQUENTIAL'. The default is 'SEQUENTIAL'.

`FORM = form`

form is a character expression with the value of 'FORMATTED' or 'UNFORMATTED'. The compiler accepts a value for this specifier, but because HP FORTRAN allows a file to contain both formatted and unformatted records, this specifier has no effect other than setting a file attribute that can be returned by subsequent INQUIRE statements.

`RECL = rcl`

rcl is an integer expression that specifies the length in bytes of each record in the file. *rcl* must be greater than zero. The default is 132.

`BLANK = blank`

blank is a character expression with the value of 'NULL' or 'ZERO'. For more information, see Considerations on page 7-72. The default is 'NULL'.

`TIMED = time`

time is a logical expression that controls timed I/O for the specified file. For more information, see Considerations on page 7-72.

`SPACECONTROL = spc`

spc is a character expression with the value 'YES', 'NO', or 'DEVICE'; it specifies whether the first character of an output record controls vertical spacing for an output device or is a data character. The default is 'DEVICE'.

`SYNCDEPTH = sync`

sync is an integer expression that specifies the number of nonretryable I/O operations your FORTRAN process can execute before it does a checkpoint. The Guardian file system or other server process must save the replies to your program's *sync* most recent requests to support your program if your program runs as a NonStop process.

The default value for *sync* is device dependent. For more information, see the FILE_OPEN_ procedure in the *Guardian Procedure Calls Reference Manual*.

`MODE = mode`

mode is a character expression with the value 'INPUT', 'OUTPUT', or 'I-O' that specifies whether the file is to be used for reading, writing, or reading and writing. The default value (in the absence of a UNIT directive or ASSIGN command specifying otherwise) is 'I-O'.

`PROTECT = prtct`

prtct is a character expression with the value 'SHARED', 'PROTECTED', or 'EXCLUSIVE' that specifies how the file is to be shared. The default value is 'SHARED'. For more information, see Considerations and the *Guardian Programmer's Guide*.

`STACK = stack`

stack is a character expression with the value 'YES' or 'NO'. For additional information, see Considerations.

Considerations

- An OPEN statement must contain one *unit* specifier and not more than one each of the other control specifiers.
- If you omit the UNIT keyword from the *unit* specifier, unit must be the first item in the list.
- You can change the value of the *blank* specifier for an open file by executing an OPEN statement that specifies BLANK = *blank*. You cannot change the value of any other *open-spec* for a unit that is already open.
- All specifiers that are quoted values (for example, 'YES') must be in upper case letters.
- Using OPEN with EDIT format files

  The file does not have to exist before you open it, but if it does not exist, you must have specified file code 101 for the file in a UNIT directive or an ASSIGN command. For information on how to create an EDIT format file, see Section 5, Introduction to File I/O in the HP NonStop Environment. The SYNCDEPTH and TIMED specifiers do not apply to EDIT format files.

  Specifying PROTECT='SHARED' when a process can have the file open with MODE='OUTPUT' or 'I-O' can cause inconsistent results.

  If you specify the ENV COMMON and NONSTOP compiler directives, the FORTRAN run-time system reports error 257 if you specify MODE = 'OUTPUT' or MODE = 'I-O'.

- Improving program performance

  If a file is a structured disk file (relative, entry-sequenced, or key-sequenced), your FORTRAN program runs faster if you specify the following attributes in the file's OPEN statement. These attributes enable sequential block buffering:

  ```
  ACCESS = 'SEQUENTIAL'
  MODE = 'INPUT'
  PROTECT = 'PROTECTED' or 'EXCLUSIVE'
  ```

- Spooler output with ENV OLD

  By default, your program uses level-3 spooling if the file you write to is a spooler collector. If you open a spooler collector for which you want to specify level-2 or level-3 spooling parameters and your program does not specify ENV COMMON, you must call the FORTRANSPOOLSTART routine after the OPEN statement and before the first WRITE statement for the file. For more information about FORTRANSPOOLSTART, see the FORTRANSPOOLSTART Routine on page 15-16.

- Spooler output with ENV COMMON

  By default, your program uses level-3 spooling if the file you write to is a spooler collector. If you want to open a spooler collector and specify level-2 or level-3 spooling parameters, and your program specifies ENV COMMON, you must call the FORTRAN_SPOOL_OPEN_ routine. FORTRAN_SPOOL_OPEN_ combines the functionality of both the OPEN statement and the FORTRANSPOOLSTART routine that you use in programs that do not specify ENV COMMON. For more information about FORTRAN_SPOOL_OPEN_, see the FORTRAN_SPOOL_OPEN_ Routine on page 15-11.

- FILE specifier

  fn must be a valid file name. If *fn* is all blanks, or if you omit this specifier and you do not supply a file name using a UNIT directive or ASSIGN command for the *unit*, FORTRAN creates a temporary file, which is deleted if the number of connections to the file goes to zero, or, if the file hasn't been closed already, then when your job ends. The temporary file is deleted regardless of the status specified when you execute an OPEN or CLOSE statement.

  If ENV COMMON is in effect, *fn* can reference a device on another node, even if the device name on the node contains eight characters, as in the following:

  ```
  OPEN (8, FILE = \NODENAM.$USERVOL.ASUBVOL.AFILE)
  ```

  You can access a volume on another node, even if the volume name contains an eight-character volume name, only if the node that you access is running a D-series system. If, when you run your program, \NODENAM is running a C-series operating system, your request will fail, even if the node on which your program is running is using a D-series system.

- Sharing access to unit 5 and unit 6

If you compile your program with ENV COMMON, your program can share access to unit 5 and unit 6 with modules written in languages other than FORTRAN. Your FORTRAN routines can share access to standard input (accessed through unit 5) and standard output (accessed through unit 6) with routines in your process written in other languages if:

- The file name for the unit is the same as the file name specified in—or defaults to—the name in the TACL IN (unit 5) or OUT (unit 6) *run-option* when you run your program.
- MODE is 'INPUT' for unit 5, 'OUTPUT' for unit 6.
- STATUS is 'UNKNOWN' or not specified.
- SYNCDEPTH is 0, 1, or unspecified.
- ACCESS is 'SEQUENTIAL' or unspecified.
- The file name is not all blanks.
- The device type is either not yet known or, if known, is a process, a terminal, a printer (if unit 6), or an unstructured or entry-sequenced disk file.
- PROTECT matches the PROTECT specifier of all previous opens to the same shared file by routines written in other languages in your process. If you do not specify the PROTECT specifier, its default is the value specified on a previous OPEN, or is device dependent if you do not specify it and this is the first open of the specified file (*fn*).

The preceding values are the defaults for each of the OPEN option specifiers except MODE, for which the default value is 'I-O'. Therefore, you must specify an appropriate value for MODE when you open a file that you want to share. If you do not specify MODE when you open unit 5 or unit 6 and you do not specify it in a TACL ASSIGN command or a FORTRAN UNIT directive, your FORTRAN routines will not take advantage of the file sharing features of the COMMON environment.

- STATUS specifier

  The default is 'UNKNOWN'.

  If you specify 'OLD', the file must exist; if you specify 'NEW' the file must not exist. Successful execution of an OPEN statement with STATUS= 'NEW' creates the specified file and changes its status to 'OLD'.

  If you specify 'SCRATCH' and a file name (FILE = *fn*), *fn* can contain only the names of a node and volume. You cannot specify a subvolume name or a file id if you specify STATUS = 'SCRATCH'. The unit is connected to a file with a system assigned temporary name for the duration of the program run or until the program closes that *unit*, at which time the file is deleted.

  If you specify 'UNKNOWN', and the file exists, it is connected to the specified *unit*. If the file does not exist, it is created and connected. In either case, the status of the file becomes 'OLD'.

- ACCESS specifier

  The default setting is 'SEQUENTIAL'.

  If you open a file with ACCESS = 'SEQUENTIAL' specified or assumed:

  - An INQUIRE statement’s ACCESS specifier returns 'SEQUENTIAL'.
  - For an EDIT format file, an ENDFILE statement or a non-update WRITE statement deletes all records following the current position.
  - For a structured (relative, entry-sequenced, or key-sequenced) disk file, READ statements use sequential block buffering if the OPEN statement also specifies
  - MODE = 'INPUT' and PROTECT = 'PROTECTED' or 'EXCLUSIVE' for the file.
  - You can still use direct or keyed access on the file, if the file type permits these.

  If you specify ACCESS = 'DIRECT' when you open a file:

  - The OPEN statement does not detect an error if the file is not capable of direct access. An error is detected only when you try to read or write the file with direct or keyed access.
  - An INQUIRE statement’s ACCESS specifier returns 'DIRECT'.
  - An ENDFILE statement has no effect for the file.
  - READ statements do not use sequential block buffering.
  - You can still access the file sequentially.

- RECL specifier

  The RECL specifier establishes the number of bytes per record in the file that you are opening. FORTRAN determines the number of bytes per record as follows:

  - FORTRAN uses *rcl* if you specify it in the OPEN statement.
  - If you do not specify *rcl* and the file is a structured disk file, FORTRAN uses the maximum length specified when the file was created.
  - If the file is not a structured disk file, FORTRAN uses the value you specify for the REC parameter in a TACL ASSIGN command.
  - If you do not specify the record length in a TACL ASSIGN command, FORTRAN uses the value you specify for the REC parameter in a UNIT compiler directive for the specified unit.
  - If the file is not a disk file, FORTRAN uses the configured record length for the device you are accessing.
  - If none of the preceding items apply, FORTRAN uses 132 bytes as the default for RECL.

- BLANK specifier

The *blank* specifier is meaningful only when you use formatted I/O on a file. The default value for the *blank* specifier is 'NULL'.

If you specify 'NULL', all blank characters read into formatted numeric input fields are ignored; a field of all blanks is treated as zero. If you specify 'ZERO', all blanks except leading blanks are treated as zero.

If you open a file that is already connected to the specified unit, only the BLANK specifier can be different from the original open.

- TIMED specifier

  .FALSE. is the default value.

  If *time* is .TRUE., timed I/O operations are allowed for the file and a TIMEOUT specifier is required in all reads and writes to the file. If *time* is .FALSE., or if you omit this specifier, the compiler ignores a TIMEOUT specifier in any I/O statement that refers to the file.

  Your program uses level-1—unbuffered—spooling if you specify TIMED = .TRUE. and the file you open is a spooler collector.

- SPACECONTROL specifier

  If you specify 'YES', the system interprets the first character of an output record as a vertical spacing control character for an output device according to the values shown in the following table:

Character	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

  If you specify 'NO', the system interprets the first character of an output record as data.

  If you specify 'DEVICE', the handling of a record depends on its destination. If it is being sent to a print device, terminal, or process, the first character is used to control vertical spacing; otherwise the whole record is assumed to be data.

- SYNCDEPTH specifier

  To enable recovery from path failure during a write, sync must be greater than zero. The default value is 1.

  For NonStop processes, the SYNCDEPTH specifier also specifies the maximum number of write operations you can safely issue to the file without executing a CHECKPOINT statement to your backup process. For additional information about checkpointing and NonStop processes, see Section 14, Interprocess Communication.

- PROTECT specifier

  When a process opens a Guardian file, the PROTECT option specifies to what degree the process is willing to share access to the file with other openers or potential openers of the file. The other openers might be in other processes or in the same process as the current opener.

  For example, if a process specifies PROTECT = 'EXCLUSIVE', the file must be closed when the process opens the file. Furthermore, any subsequent attempt to open the same file, including by the process that already has the file open, will fail until the original process closes the file.

  On the other hand, two processes can open the same file if both specify PROTECT = 'SHARED'.

  The default value is 'SHARED'.

  - 'SHARED' specifies that multiple openers can access the file for both input and output.
  - 'PROTECTED' specifies that the file can be opened for output by only one opener. Other openers can only read from the file.
  - 'EXCLUSIVE' specifies that the file can be opened by only one opener. Attempts to establish more than one open fail. Attempts to open a file with exclusive access when the file is already open also fail.

  Table 7-8 shows whether your attempt to open a file succeeds or fails, based on the attribute value:

  - You specify for the:
    - MODE option ('INPUT', 'OUTPUT', or 'I-O')
    - PROTECT option ('SHARED', 'PROTECTED', or 'EXCLUSIVE')
  - Specified by other current openers of the same file for the:
    - MODE option ('INPUT', 'OUTPUT', or 'I-O')
    - PROTECT option ('SHARED', 'PROTECTED', or 'EXCLUSIVE')

Table 7-8. File Protection and Mode Interaction Between Opening Processes (page 1 of 2)

Open Operation Attempted With		File Already Open With								
		SHARED			PROTECTED			EXCLUSIVE		
Protection Specifier	**Mode Specifier**	**I**	**O**	**I-O**	**I**	**O**	**I-O**	**I**	**O**	**I-O**
SHARED	INPUT	S	S	S	S	S	S	F	F	F
	OUTPUT	S	S	S	S	S	S	F	F	F
	I-O	S	S	S	F	F	F	F	F	F

Table 7-8. File Protection and Mode Interaction Between Opening Processes (page 2 of 2)

Open Operation Attempted With		File Already Open With								
		SHARED			PROTECTED			EXCLUSIVE		
Protection Specifier	**Mode Specifier**	**I**	**O**	**I-O**	**I**	**O**	**I-O**	**I**	**O**	**I-O**
PROTECTED	INPUT	S	S	F	F	F	F	F	F	F
	OUTPUT	S	F	F	S	F	F	F	F	F
	I-O	S	F	F	F	F	F	F	F	F
EXCLUSIVE	INPUT	F	F	F	F	F	F	F	F	F
	OUTPUT	F	F	F	F	F	F	F	F	F
	I-O	F	F	F	F	F	F	F	F	F

S means that the open is successful; F means that the open fails.

If your program is compiled with ENV COMMON in effect, and you open unit 5 or unit 6 without specifying a PROTECT attribute, the FORTRAN run-time library determines the protection attribute based on the type of device you are accessing and the protection attribute of any previous opens of the same unit.

Note. If you compile your program with either a C-series FORTRAN compiler or a D-series FORTRAN compiler for which you specify ENV OLD, and you open unit 5 or unit 6, the FORTRAN run-time routines use 'SHARED' as the value for the PROTECT attribute if you do not specify one.

If you compile your program with a D-series FORTRAN compiler, you specify ENV COMMON, and you open unit 5 or unit 6 without specifying the PROTECT attribute, FORTRAN determines the PROTECT attribute as follows:

- If the file is already open, FORTRAN uses the PROTECT attribute specified when the file was first opened.
- If the file is not already open, FORTRAN determines the PROTECT attribute based on the type of device you are opening. This table shows the default protection for the devices you can open for units 5 and 6:

Device	**Unit 5**	**Unit 6**
Process	'PROTECTED'	'EXCLUSIVE'
$RECEIVE	'EXCLUSIVE‘	N.A.
Disk	'PROTECTED'	'EXCLUSIVE'
Printer	N.A.	'EXCLUSIVE'
Terminal	'SHARED’	'SHARED'

- STACK checkpoint specifier

If you run your program as a NonStop process, an OPEN statement automatically checkpoints program environment information to your backup process. The effect of this specifier is the same as that in the CHECKPOINT statement, which is described in this section and in Section 14, Interprocess Communication.

- Error conditions

  If you specify *label*, and an error occurs while FORTRAN is opening the file, the OPEN statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by label. If you also specified ios, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *label*, and an error occurs on the OPEN statement, your program continues executing with the statement that follows the OPEN statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *label*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
OPEN (60, FILE='outfile',STATUS='NEW',IOSTAT=misopen)
OPEN (UNIT=1, FILE='$receive',MODE='INPUT',RECL=346)
OPEN (UNIT=master,FILE='mastrprt', SYNCDEPTH=1)
```

PARAMETER Statement

The PARAMETER statement assigns a symbolic name to a constant value.

```
PARAMETER ( name = exp [, name = exp ]... )
```

name

is a symbolic name.

exp

is a constant expression.

Considerations

- The PARAMETER statement defines the value of each *name* by evaluating its corresponding expression.
  - If the first letter of *name* designates a numeric type, *exp* must be an arithmetic expression.
  - If the first letter of *name* designates a character type, *exp* must be a character expression.

- If the first letter of name designates a logical type, *exp* must be a logical expression.

- If a symbolic name appears in the constant expression, it must have been previously defined in the same or a different PARAMETER statement in the same program unit.
- If the implicit data type of *name* is not the correct type for *exp*, you must specify the type of *name* in a type statement or an IMPLICIT statement before you use *name* in a PARAMETER statement.

  If *exp* is a character constant that has more than one character, you must specify the number of characters in name before using it in a PARAMETER statement.
- Once you have defined a symbolic name with the PARAMETER statement, it can only identify its corresponding constant in that program unit.
- You cannot use the symbolic name of a constant in a format specification.
- You cannot use the symbolic name of a constant to form part of another constant, for example either part of a complex constant.
- You cannot change the type or length of the symbolic name of a constant in subsequent statements.
- You cannot define the symbolic name of a constant more than once in a program unit.

Examples

Numeric type:

```
REAL fees, student body dues
PARAMETER (fees = 207.50, student body dues = 15.33)
REAL tuition
PARAMETER (tuition = fees + student body dues)
total fees = tuition * credits
```

Character type:

```
CHARACTER title*20
PARAMETER (title = 'Engineering Report')
```

Logical type:

```
LOGICAL yes, no
PARAMETER (yes = .TRUE., no = .FALSE.)
```

PAUSE Statement

The PAUSE statement temporarily halts program execution.

```
PAUSE [ message ]
```

message

is an unsigned integer constant of up to five digits or a character constant of up to 80 characters that is displayed when the program executes the PAUSE statement.

If you compile your program with ENV OLD in effect, FORTRAN displays *message* on your home terminal unless you specify the TERM run-option when you run your program, in which case FORTRAN displays *message* on the device you specify in the TERM run-option.

If you compile your program with ENV COMMON, FORTRAN writes *message* to the standard log file. For more information about the standard log file, see the *CRE Programmer's Guide*.

Considerations

- The PAUSE statement temporarily halts execution of the program in which it occurs, until you enter a carriage return via the home terminal. Program execution resumes with the next executable statement following the PAUSE statement. The program ignores any data entered at the home terminal while in the paused state.
- message enables you to display a numeric identifier or a text message when the PAUSE statement executes. *message* is optional; if you do not specify *message*, FORTRAN displays the single word "PAUSE".

Examples

```
PAUSE 55566

PAUSE 'Mount next tape!'
```

POSITION Statement

The POSITION statement makes it possible for sequential-access I/O statements to perform random access of structured files, either by record number or by specified primary or alternate keys.

```
POSITION ( [UNIT =] unit [, IOSTAT = ios ]
  [, ERR = lbl ], position)
```

unit

is an integer expression from 1 through 999 that specifies the unit to be positioned.

ios

is a variable or array element of integer type that returns an error number or zero (no error) following execution of the POSITION statement. For more information about error numbers, see the Error Numbers on page 6-5.

lbl

is an integer expression that designates the label of an executable statement in the same program unit to which control passes if an I/O error occurs during positioning.

position

is a position specifier that assumes one of the following, mutually exclusive, forms:

```
REC = recno
```

or

```
KEY= key, KEYLEN= exp, KEYID= kid, MODE= mode
[, COMPARELEN= clen] [, SKIPEXACT= skip]
```

recno

is an expression of INTEGER*4 type that specifies the record number to which the file is to be positioned. Note that the first record in a file is record number one.

key

is a character expression whose value is a primary or alternate key value.

exp

is an integer expression that specifies the number of bytes in the key field to compare to the same number of bytes in *key*.

kid

is an integer or character expression whose value specifies which key to use. See Considerations on page 7-83 The default value is 0.

mode

is a character expression whose value is either 'APPROXIMATE', 'GENERIC', or 'EXACT'. The default value is 'APPROXIMATE'.

clen

is an integer expression whose value is in the range 0 through 255. The default value is 0.

skip

is a character expression whose value is either 'YES' or 'NO'. The default value is 'NO'.

Considerations

- A POSITION statement establishes the record to access on the next READ or WRITE statement. The POSITION statement does not perform I/O operations, nor does it check for the existence of the specified record. The file is positioned to the specified record when your program executes the next data transfer statement. For more information about positioning structured files, see the *ENSCRIBE Programmer's Guide*.
- You can code the control specifiers in any order, except that if you omit the UNIT keyword from the unit specifier, the UNIT specifier must be the first item in the list.
- If you use the KEY specifier, you must also specify values for KEYLEN, KEYID, and MODE. The file must be a relative, entry-sequenced, or key-sequenced file. The key can be an alternate key for any of the allowed file types, or the primary key of a key-sequenced file. For additional information, see Section 5, Introduction to File I/O in the HP NonStop Environment.
- KEYLEN specifier

  If you omit the KEYLEN specifier, the compiler uses the actual length of the *key* field.
- KEYID specifier

  If you use an alternate key to position the file, *kid* must be a character expression having a value equal to the two-character ENSCRIBE key tag defined for the alternate key.

  If you use the primary key to position the file, *kid* must be either 0 or blank.

  If you omit this specifier, the compiler assumes a value of 0.
- MODE specifier

  If you specify 'APPROXIMATE', ENSCRIBE uses the specified *key* as the full or partial value of the key that starts the sequential reading of the file. The system detects an end-of-file condition when it reaches the end of the file.

  If you specify 'GENERIC', ENSCRIBE uses the specified *key* as the full or partial value of the key that starts the sequential reading of the file. The system detects an end-of-file condition when the record key no longer matches *key* for the specified key length. 'EXACT' is the same as 'GENERIC', except that ENSCRIBE interprets *key* as the full value of the key to be used.

  The default value is 'APPROXIMATE'.
- COMPARELEN specifier

When performing a GENERIC file search on an alternate key, the record made available to the program is the first one that satisfies the search requirements. If you want a later record, you can use the COMPARELEN specifier to include characters of the primary key in the alternate key comparison. For example, if you are searching for the alternate key 'RIVET', and it is known that part numbers (the primary key) in the 12100 range refer to 1-inch rivets, the 12200 range to 2-inch rivets, and so on, a key value of RIVET 122 with a COMPARELEN=8 specifier can concatenate the first three characters of the primary key with the 5-character alternate key to narrow the search. The value you specify with COMPARELEN is therefore the sum of the full length of the key plus additional characters from the primary key.

Do not use this specifier if you are performing an EXACT search or using the primary key.

If you omit this specifier, the compiler assumes a value of 0 for COMPARELEN.

- SKIPEXACT specifier

  If you are using an alternate key to perform a file search, the first record found that specifies the search requirements is made available to the program. If you save the key values of that record and use them in a subsequent search, the same record is found. If you specify SKIPEXACT='YES', the record that exactly matches the previously found record is skipped and the next record with the same alternate key is made available.

  If you omit this specifier, the compiler assumes a value of 'NO' for the SKIPEXACT specifier.

  Note that you can use this specifier with primary keys thus allowing a FORTRAN server to read the next record in a file and remain context free.

- Direct Access

  You can use the REC specifier only with EDIT format files, unstructured files, and relative disk files. In all of these file types, records are identified by INTEGER*4 record number values. You do not need to use a POSITION statement to position direct access files. The compiler treats any READ or WRITE statement containing a REC= specifier as a POSITION statement followed by a sequential READ or WRITE statement.

  If you do use a POSITION statement, a subsequent READ statement reads the record with the smallest record number that is greater than or equal to the specified record number. A WRITE statement inserts or replaces a record with the specified record number.

  To append new lines at the end of an EDIT format file, OPEN the file and execute a POSITION or WRITE statement, with REC= -2.

  You cannot POSITION an EDIT format file in a FORTRAN program that is running as a NonStop process.

- If you position unit 5 or unit 6 and you have not already established a connection for the unit, POSITION implicitly opens the unit using default parameters. If you specify ENV COMMON and you position unit 5 or unit 6, your FORTRAN routines share access to standard input or standard output, respectively, with routines written in other languages only if the access mode for unit 5 is INPUT and for unit 6 is OUTPUT. However, the default access mode for both units 5 and 6 is I-O. If you want to share access to the file connected to the unit, you must set the unit's access mode to INPUT (unit 5) or OUTPUT (unit 6) before you execute the POSITION statement. You can set the access mode:

  - In a FORTRAN OPEN statement, as in

    ```
    OPEN( 5, MODE = 'INPUT')
    OPEN( 6, MODE = 'OUTPUT')
    ```

  - In a TACL ASSIGN command, as in

    ```
    ASSIGN FT005, , INPUT
    ASSIGN FT006, , OUTPUT
    ```

  - In a UNIT compiler directive, as in

    ```
    UNIT (5, INPUT)
    UNIT (6, OUTPUT)
    ```

  For more information about using units 5 and 6 as shared files, see the OPEN Statement on page 7-70.

- If a POSITION statement causes unit 5 or unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.

- Error conditions

  If you specify *lbl*, and an error occurs during the position operation, the POSITION statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by *lbl*. If you also specify *ios*, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *lbl*, and an error occurs during the position operation, your program continues executing with the statement that follows the POSITION statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
POSITION(UNIT=100, IOSTAT=inerror, REC=inrecnum)
POSITION(8,ERR=30,KEY=partnum,KEYID=0,KEYLEN=5,MODE='EXACT')
POSITION(UNIT=12, KEY=pname, KEYID='PN', KEYLEN=20,
& MODE='GENERIC', COMPARELEN=8, SKIPEXACT='YES')
```

PRINT Statement

The PRINT statement writes data to the preconnected output unit (unit 6) only.

```
PRINT format [, output-list ]
```

format

is one of the following:

- The label of a FORMAT statement in the same program unit
- An integer variable in which an ASSIGN statement has stored the label of a FORMAT statement
- The name of a character array that contains a format specification
- A character expression that yields a format specification
- An asterisk indicating list-directed formatting

output-list

is a list of entries, separated by commas; an entry can be any valid expression except a character expression involving concatenation of a dummy argument having a length specification of (*). An entry can also take the form of an implied DO list. See Using Implied DO Lists on page 5-27.

Considerations

- A PRINT statement writes data from its output list to unit 6. If you have not already established a connection for unit 6, a PRINT statement implicitly opens unit 6 using default parameters.
- If a PRINT statement causes unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.
- If you specify ENV COMMON, your FORTRAN routines share access to standard output with routines in your process written in languages other than FORTRAN only if the access mode for unit 6 is OUTPUT. However, the default access mode for unit 6 is I-O. If you want to share access to the file connected to unit 6 you must set the access mode for unit 6 to OUTPUT. You can set the access mode:

- In a FORTRAN OPEN statement, as in

```
OPEN( 6, MODE = 'OUTPUT')
```

- In a TACL ASSIGN command, as in

```
ASSIGN FT006, , OUTPUT
```

- In a UNIT compiler directive, as in

```
UNIT (6, OUTPUT)
```

For more information about using unit 6 as a shared file, see the OPEN Statement on page 7-70.

- The PRINT statement uses the first character of each record to control vertical spacing if you specify
  - SPACECONTROL = 'YES' when you open unit 6.
  - SPACECONTROL = 'DEVICE' when you open unit 6—explicitly or implicitly—and the device is a terminal, a printer, or a process.

  The default value for SPACECONTROL is 'DEVICE'. The following table shows how FORTRAN interprets the first character of each record printed if SPACECONTROL is 'YES' or if SPACECONTROL is 'DEVICE' and the device to which you are printing is a terminal, a printer, or a process.

Character Vertical	Spacing Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

  Note that these codes do not control vertical spacing on all devices.

- Execution of a PRINT statement for a file that does not exist creates that file (if no error condition occurs).
- An entry in the *output-list* can assume the form of an implied DO list. For example, the following statement prints every element of array EMPLOYEES:

```
PRINT 90, (employees(k), k = 1,289)
```

- You cannot PRINT an EDIT format file in a FORTRAN program that is running as a NonStop process.
- If you specify ENV OLD or you do not specify an ENV directive, your program uses the C-series FORTRAN library.

  If you specify ENV COMMON, your program uses the D-series FORTRAN runtime library. For more information, see Section 13, Mixed-Language Programming.

Examples

```
PRINT *, 'Balance Due :$', balance
PRINT 200, base squared, base cubed
```

PROGRAM Statement

The PROGRAM statement assigns a symbolic name to the main program unit.

```
PROGRAM prog-name
```

prog-name

is a valid symbolic name

Considerations

- The PROGRAM statement is optional. If you use it, it must be the first statement of the main program unit.
- *prog-name* is global to the executable program and must be different from the name of every external procedure, block data subprogram, or common block in the same executable program. *prog-name* must not be the same as any local name in the main program.

Example

The following statement assigns the name PRIMES to the main program unit.

```
PROGRAM primes
```

READ Statement

The READ statement inputs data from a specified unit or file.

```
READ { format  [,  input-list]                                  }
     { (read-spec  [,  read-spec]. . . )  [input-list]          }
```

format

is either the label of a FORMAT statement in the same program unit, the name of an integer variable in which the ASSIGN statement has stored the label of a FORMAT statement, the name of a character array that contains a format specification, a character expression that yields a format specification, or an asterisk indicating list directed formatting.

input-list

is a list of items separated by commas. A list item is either the name of a variable, an array, an array element, a character substring, a RECORD, a RECORD field, or an implied DO list.

read-spec

is one of the following control specifiers. You can write these specifiers in any order unless you omit the UNIT or FMT keywords; for more information, see Considerations on page 7-91.

[UNIT=] *unit*

unit is one of the following:

- An integer expression from 1 through 999 that corresponds to a unit previously connected by an OPEN statement or to a preconnected unit.
- An asterisk (*), which implies unit 5. Unit 5 is preconnected for formatted sequential input.
- An internal file identifier.

If you omit the UNIT keyword, *unit* must be the first entry in the list.

For information about external and internal files, see External and Internal Files on page 5-3.

[FMT=] *format*

See the format on page 7-86

If you use an asterisk to indicate list-directed formatting, you cannot use a record specifier in the control list.

If you omit the optional FMT keyword, *format* must be the second item on the control list and *unit*, without the UNIT specifier, must be the first item on the list.

REC = *rec*

rec is an expression of INTEGER*2 or INTEGER*4 type that specifies the record number of the record to read.

IOSTAT = *ios*

ios is an integer variable or integer array element in which FORTRAN returns an error number if an error occurs while executing the READ statement. If the READ statement is successful, *ios* is zero. If the READ statement encounters an end of file, *ios* is -1. For more information about error numbers, see the Error Numbers on page 6-5.

`ERR = lbl`

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while executing the READ statement.

`END = endlbl`

endlbl is an integer that specifies the label of an executable statement within the same program unit to which control passes if the READ statement encounters an end of file.

`LOCK = lock`

lock is a logical expression that specifies whether records are available to other processes.

`UPDATE = update`

update is a logical expression that indicates whether records in structured files are to be updated or deleted.

`LENGTH = len`

len is an INTEGER*2 variable that returns the actual length, in bytes, of the record most recently read.

`TIMEOUT = to`

to is an INTEGER*2 or INTEGER*4 expression that specifies the maximum time the system should wait for the read operation to complete.

`PROMPT = message`

message is a character expression that specifies the prompt message to display when input is expected.

`PROMPTLENGTH = plen`

plen is an integer expression that specifies the length of the prompt message.

`SOURCE = receive`

is an integer array of at least 16 elements that contains information about the requester. For additional details, see Section 14, Interprocess Communication.

Considerations

- Order of read specifiers

  Entries for read-spec can be in any order, except that:

  - If you omit the UNIT keyword, unit must be the first item in the list.
  - If you omit the FMT keyword, format must be the second item in the list, unit must be the first item in the list, and you must specify unit without the UNIT keyword.

- REC specifier

  The control list must not contain a record specifier if any of the following are true:

  - The file is connected for sequential access
  - The unit specifier names an internal file
  - The format specifier is an asterisk

  You cannot use a record specifier on an EDIT format file in a FORTRAN program that is running as a NonStop process.

  If the control list contains a record specifier, it must not contain an end-of-file specifier.

- IOSTAT specifier

  See the Error conditions on page 7-93 and End-of-file condition on page 7-93.

- ERR specifier

  See the Error conditions on page 7-93.

- END specifier

  See the End-of-file condition on page 7-93.

- LOCK specifier

  The default value is .FALSE.. The LOCK specifier is relevant only for structured files.

  The lock specifier coordinates access to a file shared by two or more processes. If *lock* is .TRUE., the current record becomes temporarily unavailable to other processes. Use a subsequent WRITE statement with *lock* set to .FALSE. to restore record access to other processes sharing the file.

- UPDATE specifier

  The default value is .FALSE.. This specifier is relevant only for disk files (except EDIT format files) and processes.

  If the value of *update* is .TRUE., FORTRAN uses the READUPDATE or READUPDATELOCK procedure; if the value is .FALSE., FORTRAN uses the

READ or READLOCK procedure. For additional information about these procedures, see the *Guardian Procedure Calls Reference Manual*.

- LENGTH specifier

  Use the LENGTH specifier to determine the actual length of variable-length records. If the READ statement terminates, *len* returns the actual length, in bytes, of the record most recently read.

  Note that the amount of the input record actually read may depend on format control; *len* returns the full length of the record, whether or not all the record is actually returned to your program.

- TIMEOUT specifier

  to represents a number of hundredths of a second. If the READ operation has not completed when this time has elapsed, statement execution terminates with file management error 40.

  You can use the TIMEOUT specifier to impose a time limit only if you specify TIMED=.TRUE. when you open the file.

- PROMPT and PROMPTLENGTH specifiers

  These specifiers are relevant only if the file is a terminal or a process.

  The first character of the prompt has the effect defined by the SPACECONTROL specifier of the OPEN statement.

  You do not need to include a PROMPTLENGTH specifier. If you omit *plen*, the actual number of characters defined for *msg* is output.

- Implied DO list

  See Using Implied DO Lists on page 5-27.

- Reading from unit 5

  If you read a record from unit 5 and you have not already established a connection for the unit, READ implicitly opens the unit using default parameters. If you specify ENV COMMON and you read a record from unit 5, your FORTRAN routines share access to standard input with routines written in other languages only if the access mode for the unit is INPUT. However, the default access mode for unit 5 is I-O. If you want to share access to the file connected to unit 5, you must set the unit's access mode to INPUT before you execute the READ statement. You can set the access mode:

  - In a FORTRAN OPEN statement, as in

    ```
    OPEN( 5, MODE = 'INPUT')
    ```

  - In a TACL ASSIGN command, as in

    ```
    ASSIGN FT005, , INPUT
    ```

- In a UNIT compiler directive, as in

  ```
  UNIT (5, INPUT)
  ```

  For more information about using unit 5 as a shared file, see the OPEN Statement on page 7-70.

- If a READ statement causes unit 5 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.

- End-of-file condition

  If you specify *endlbl*, and the read operation encounters an end of file, the READ statement terminates and FORTRAN transfers control to the statement identified by *endlbl*. If you also specify *ios*, it will have a value of -1.

  If you specify *ios*, but not *endlbl*, and the read operation encounters an end of file, your program continues executing with the statement that follows the READ statement. You can analyze *ios* to determine the error that occurred. *ios* will have a value of -1 for an end-of-file condition.

  If you do not specify *ios* or *endlbl*, and the read operation encounters an end of file, FORTRAN terminates your program and displays a run-time diagnostic message.

- Error conditions

  If you specify *lbl*, and an error occurs during the read operation, the READ statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by *lbl*. If you also specify *ios*, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *lbl*, and an error occurs during the read operation, your program continues executing with the statement that follows the READ statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
READ (4,*) firstname, age
READ (UNIT=4, PROMPT= ' Enter part no. ',FMT=*) partnum
READ (infile, 400, ERR=30, END=500, REC=numrec)
+    ( array(k), k=1, stop)
```

RECORD Statement

The RECORD statement defines a data structure, which can include data of different types.

```
RECORD record-name [ ( [ lower:] upper) ]
  [ field-declaration ]...
END RECORD
```

record-name

is a symbolic name or array declarator

field-declaration

is either a data type declaration, an EQUIVALENCE statement, a record declaration, or a FILLER * *nnn* where *nnn* is an unsigned integer constant in the range of 1 through 255.

lower

is an integer expression that specifies the lower bound of a one-dimensional array.

upper

is an integer expression that specifies the upper bound of a one-dimensional array.

Considerations

- Observe the following restrictions in using the RECORD statement:
  - Start a RECORD declaration with the RECORD statement and end it with the END RECORD statement.
  - You can nest RECORD declarations, up to 15 deep.
  - You can use an array in a RECORD or sub-RECORD, but the array can have only one dimension.
  - You can declare a RECORD's dimensions using a COMMON, DIMENSION, or RECORD statement; but you can declare the RECORD's dimensions only once in a program unit.
  - You can equivalence RECORDs only to other RECORDs.
  - You cannot use a DATA statement to initialize RECORDs.
  - You cannot declare RECORDs in a BLOCK DATA subprogram.
- For additional information about the RECORD statement, see the Records on page 2-20.

Example

```
RECORD employees
   FILLER*19
   RECORD address
      CHARACTER*20 street
      CHARACTER*10 city
      CHARACTER*5 state
      INTEGER zip
   END RECORD
   RECORD grade
      CHARACTER*10 department
      REAL pay
      INTEGER*4 empnumber
   END RECORD
END RECORD
```

RETURN Statement

The RETURN statement terminates execution of a subprogram and returns control to the calling program unit.

```
RETURN [ iexp ]
```

iexp

is an integer expression that designates an alternate return from the subroutine.

Considerations

- A procedure subprogram ends with an END statement. However, both function subprograms and subroutine subprograms can include one or more RETURN statements to designate alternate exit points. For example:

```
SUBROUTINE numbers (j, k, n)
.
READ (*,*) number
IF (number .GT. 1) RETURN
.
END
```

- A RETURN statement without an *iexp* entry returns control to the first executable statement that follows the statement which called it. The *iexp* entry, which must be an integer expression, specifies that control return to the statement label specified in the calling statement that corresponds to the position of the alternate return expression in the actual parameter list of the calling program unit.
- If FORTRAN cannot execute an alternate return because the value for *iexp* is less than one or greater than the number of dummy arguments, control passes to the first executable statement after the CALL statement.
- Executing a RETURN statement terminates the association of actual arguments with dummy arguments in a subprogram. For information about how to preserve associated values after exiting a subprogram, see the SAVE Statement on page 7-99.

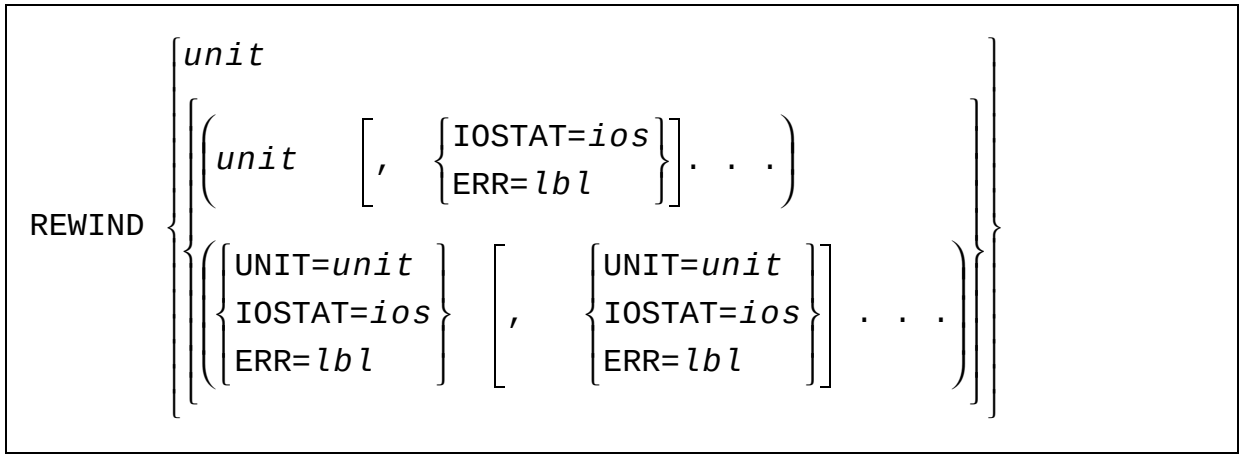
Example

In the following example, control passes to statement 100 if the first RETURN statement executes, and to statement 230 if the second RETURN statement executes:

```
PROGRAM MAIN
CALL products (price, tax, *100, *230)
.
END

SUBROUTINE products (p, t, *, *)
.
RETURN 1
.
RETURN 2
.
END
```

REWIND Statement

The REWIND statement positions a file connected to a specified unit at its initial point. If the file is already at its initial point, or if the file is connected but does not exist, the REWIND statement has no effect.



unit

is an integer expression from 1 through 999 that identifies an external unit connected for sequential access. The unit must be connected to a magnetic tape, a process, an unstructured file with fixed-length records, a relative file, an EDIT format file or an entry-sequenced file.

ios

ios is a variable or array element of integer type that returns an error number or zero (no error) following the REWIND operation. For more information about error numbers, see the Error Numbers on page 6-5.

lbl

lbl is the label of an executable statement in the same program unit to which control passes if an error occurs during a rewind operation.

Considerations

- If you rewind unit 5 or unit 6 and you have not already established a connection for the unit, REWIND implicitly opens the unit using default parameters. If you specify ENV COMMON and you rewind unit 5 or unit 6, your FORTRAN routines share access to standard input or standard output, respectively, with routines written in other languages only if the access mode for unit 5 is INPUT and for unit 6 is OUTPUT. However, the default access mode for both units 5 and 6 is I-O. If you want to share access to the file connected to the unit, you must set the unit's access mode to INPUT (unit 5) or OUTPUT (unit 6) before you execute the REWIND statement. You can set the access mode:

- In a FORTRAN OPEN statement, as in

```
OPEN( 5, MODE = 'INPUT')
OPEN( 6, MODE = 'OUTPUT')
```

- In a TACL ASSIGN command, as in

```
ASSIGN FT005, , INPUT
ASSIGN FT006, , OUTPUT
```

- In a UNIT compiler directive, as in

```
UNIT (5, INPUT)
UNIT (6, OUTPUT)
```

For more information about using units 5 and 6 as shared files, see the OPEN Statement on page 7-70.

- If a REWIND statement causes unit 5 or unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.
- Error conditions

  If you specify *lbl*, and an error occurs during the rewind operation, the REWIND statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by *lbl*. If you also specify *ios*, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *lbl*, and an error occurs during the rewind operation, your program continues executing with the statement that follows the REWIND statement. You can analyze *ios* to determine the error that occurred, if any.

  If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.
- Use of REWIND with EDIT format files

  After a REWIND statement, the NEXTREC specifier of an INQUIRE statement returns -1 as the latest number of an EDIT format file.

Examples

```
REWIND 123
REWIND (ERR=370, UNIT=2)
```

SAVE Statement

The SAVE statement saves the status of specified entities after the termination of a subprogram.

```
SAVE [ name [, name ]... ]
```

name

is a common block name enclosed in slashes, a variable name, array name, or RECORD name.

Considerations

- All entities defined in a subprogram unit become undefined when a RETURN or END statement executes except for the following:
  - Entities named in a SAVE statement
  - Entities in any common block
  - Entities declared in a DATA statement
- If the SAVE statement specifies an entity local to the subprogram, FORTRAN saves the current value of that entity when the subprogram terminates and makes it available to the subprogram the next time it executes.
- If you omit the name list, the SAVE statement saves all the allowable entities defined by the subprogram.
- The SAVE statement cannot include dummy argument names, procedure names, or names of entities in common blocks.
- To save anything in a common block, you must save everything in it by specifying the common block name as follows:

  ```
  SAVE /customers/
  ```

  Saving common blocks has no effect in HP FORTRAN because all common blocks are always saved, but SAVE is supported for ANSI standard conformance.

Example

```
SAVE personnel, payroll, /employee/
```

START BACKUP Statement

The START BACKUP statement defines control options for fault-tolerant processing. It also starts the backup process, ensures that the backup process opens all files currently open in the primary process, checks file synchronization information, and checkpoints all usable memory. START BACKUP is an extension to the ANSI

standard and gives the FORTRAN user access to the HP fault-tolerant programming facility.

```
START BACKUP [ ( start-spec [, start-spec ]... ) ]
```

start-spec

is one of the following specifiers:

CPU = *number*

number is an integer expression with a value from -1 through 15 that specifies in which processor to start the backup process. The default value is 0. If you specify CPU = -1, the system determines the processor in which the backup process runs.

ERR = *label*

label is an integer that designates an executable statement, in the same program unit, to which control passes if an error occurs during an attempt to start a backup process.

OPTION = *int*

int is an integer expression that controls a number of options for NonStop process. Table 7-9 on page 7-101 shows the meaning of each OPTION bit.

BACKUPSTATUS = *var*

var is an integer variable that returns zero (no error) or an integer that identifies the error following execution of a START BACKUP statement. Table 7-10 on page 7-102 shows the status codes that FORTRAN returns to your program.

Considerations

- The START BACKUP statement does not establish a takeover point. You must execute a CHECKPOINT statement to establish a takeover point.
- You can write the *start-specs* in any order, but you cannot specify any item more than once.
- For additional information about fault-tolerant programming, see Section 16, Fault-Tolerant Programming.

Table 7-9. Option Bits for START BACKUP OPTION Specifier

Bit	Option
0-5	Reserved; must be zero.
6	If ENV OLD, bit 6 must be zero. If ENV COMMON, bit 6 specifies whether START BACKUP should checkpoint the extended stack when a backup process is created. 0 Do not checkpoint when starting the backup. 1 Checkpoint when starting the backup.
7	Specifies whether to maintain an outstanding NOWAIT read operation on $RECEIVE: 0 Maintain a read. 1 Do not maintain a read.
8	Reserved; must be zero.
9	Specifies FORTRAN action if your program successfully executes an OPEN statement but the system cannot open the same file in the backup process: 0 The primary closes the file. 1 The primary stops the backup process. For more information, see the “CHECKOPEN Procedure” if you compile your program with ENV OLD, or the “FILE_OPEN_CHKPT_” procedure if you compile your program with ENV COMMON, both in the *Guardian Procedure Calls Reference Manual*.
10	Specifies how FORTRAN handles a “bad checkpoint parameter” error on CHECKPOINT: 0 The primary ABENDs. 1 FORTRAN returns an error code to the FORTRAN program.
11	Specifies when to recreate a backup process after a takeover: 0 At the next CHECKPOINT statement. 1 Immediately.
12	Reserved; must be zero.
13	Specifies the disposition of the backup if the primary is stopped by a TACL STOP command or a programmatic call to the STOP or PROCESS_STOP_ system procedure: 0 The backup stops also. 1 The backup becomes the primary and attempts to create a new backup (this feature is useful for testing).
14	Reserved; must be zero.
15	Reserved; must be zero.

Note: Following the standard HP convention, bit positions are numbered 0 to 15 from left to right within a 16-bit word.

Table 7-10. Status Codes Returned for CHECKPOINT and START BACKUP (page 1 of 2)

Error Number	Description
0	No error.
100	Takeover by backup; primary process stopped.
101	Takeover by backup; primary process ABENDed.
102	Takeover by backup; primary processor module failed.
103	Takeover by backup; primary process called the CHECKSWITCH system procedure.
1000	Backup processor module is down.
2nnn	Communication error; *nnn* is a File System Error code.
3nnn	Failure to open a file in the backup process that is open in the primary process; nnn is a File System Error code.
4nnn	If ENV OLD, a NEWPROCESS failure. If ENV COMMON, a PROCESS_CREATE_ failure; *nnn* is a Guardian file management error, for 0 < *nnn* < 512.1
49xx	If ENV OLD, other NEWPROCESS failure. If ENV COMMON, other PROCESS_CREATE_ failure; : *xx* is[1]:

Table 7-10. Status Codes Returned for CHECKPOINT and START BACKUP (page 2 of 2)

Error Number	Description	
	ENV OLD	**ENV COMMON**
	01: Undefined externals	02: Parameter error
	02: No PCB available	03: Bounds error
	04: Unable to allocate map	04: File system error on library file
	05: Unable to get virtual disk space	05: File system error on swap file
	06: Illegal file format	06: File system error on extended swap file
	07: Unlicensed privileged program	08: Invalid home terminal
	10: Unable to communicate with System Monitor	09: I/O error to home terminal
	12: Program file and library file specified are the same file	10: Unable to communicate with system monitor
	13: Extended data segment initialization error	12: Illegal program-file format
	14: Extended segment swap file error	13: Invalid library-file format
	15: Illegal home terminal	14: Process has undefined externals
	16: I/O error to home terminal	15: No process control block available
	18: Object file with illegal process device subtype	16: Unable to allocate virtual address space
	19: Process device subtype in backup process is not the same as in primary process	17: Unlicensed privileged program
	24: Unrecognized error from remote node	18: Library conflict
		19: Program file and library file specified are the same file
		20: Object file has illegal process device subtype
		21: Process device subtype specified in backup is not the same as that in primary
		22: Backup creation was specified but caller is unnamed
		24: DEFINE error
		27: PFS size in object file is invalid
		28: Unrecognized error from remote node
5000	Too many failure/restart cycles (more than 10).	
6000	Parameter passing error, or program logic error.	

[1]For more information, see the *Guardian Procedure Calls Reference Manual*.

Examples

- The following example specifies ENV OLD:

  ```
  ?ENV OLD

     START BACKUP( CPU = 2, ERR = 400, OPTION = 16)
  ```

  A START BACKUP statement with OPTION = 16 (only bit 11 is set) specifies the following actions:

  - Maintain a permanent nowait read on $RECEIVE.
  - The primary process closes a file that it has just opened, if the backup process cannot also open the file.
  - The primary process ABENDs if CHECKPOINT causes a “bad checkpoint parameter” error.
  - A new backup is created immediately by the FORTRAN run-time system after a takeover.
  - The backup process stops if a STOP command stops the primary process.

- The following example specifies ENV COMMON.

  ```
  ?ENV COMMON

     START BACKUP( CPU = 2, ERR = 400, OPTION = %1564)
  ```

  If your program specifies ENV COMMON, OPTION = %1564 (all defined option bits set to 1) specifies:

  - The START BACKUP statement checkpoints the extended stack when a backup process is created.
  - FORTRAN does not maintain a permanent nowait read on $RECEIVE.
  - The primary process stops the backup process if the backup process cannot successfully open a file just opened by the primary process.
  - FORTRAN returns an error code to the program if CHECKPOINT causes a “bad checkpoint parameter” error.
  - A new backup is created immediately by the FORTRAN run-time library after a takeover.
  - The backup becomes the new primary and creates a new backup if a STOP TACL command stops the original primary.

STOP Statement

The STOP statement terminates program execution.

```
STOP [ message ]
```

message

is an unsigned integer constant up to five digits long or a character constant up to 80 characters long that FORTRAN displays when your program executes a STOP statement.

If you compile your program with ENV OLD in effect, FORTRAN displays *message* on your home terminal unless you specify the TERM run-option when you run your program, in which case FORTRAN displays *message* on the device you specify in the TERM run-option.

If you compile your program with ENV COMMON, FORTRAN writes *message* to the standard log file. For more information about the standard log file, see the *CRE Programmer's Guide*.

Considerations

- The STOP statement terminates execution of the entire executable program. Executing a STOP statement in a subroutine terminates execution of that subroutine, the calling main program, and all other subroutines called by the main program.
- FORTRAN automatically closes files that are open when the STOP statement executes.
- If you compile your FORTRAN program with the ENV OLD directive in effect, you can call the FORTRANCOMPLETION utility routine instead of executing a STOP statement. FORTRANCOMPLETION performs the same tasks as STOP, but you can also specify completion codes and related information for the STOP and ABEND procedures. For more information, see the FORTRANCOMPLETION Routine on page 15-2.

  If you compile your FORTRAN program with the ENV COMMON directive in effect, you can call the FORTRAN_COMPLETION_ utility routine instead of executing a STOP statement. FORTRAN_COMPLETION_ performs the same tasks as STOP, but you can also specify completion codes and related information for the PROCESS_STOP_ procedure. For more information, see the FORTRANCOMPLETION Routine on page 15-2.

Example

```
STOP 'End of job.'

STOP 55489
```

SUBROUTINE Statement

The SUBROUTINE statement is the first statement of a subroutine.

```
SUBROUTINE name [ ( [ dummy [, dummy ]... ] ) ]
```

name

is the symbolic name of the subroutine. It has the scope of an executable program.

dummy

is a variable name, an array name, a RECORD name, a dummy procedure name or an asterisk, which corresponds to an alternate return specifier in a CALL statement.

Considerations

- A subroutine must begin with a SUBROUTINE statement and end with an END statement.
- A subroutine can include one or more ENTRY statements and one or more RETURN statements.
- The name of *dummy* has the scope of the subroutine. Do not declare it in an EQUIVALENCE, PARAMETER, DATA, INTRINSIC, SAVE, or COMMON statement (except as a common block name) within the program unit.
- The subroutine name has the scope of an executable program and must be different from any other external name.
- The subroutine name must not be the same as any common block name declared anywhere in the executable program.
- For additional information about subroutines, see Section 4, Program Units.

Example

```
SUBROUTINE namelength (name, length)
   CHARACTER name*(*)
   length = LEN(name)
   IF (length .GT. 80) THEN
      PRINT *, 'YOUR ENTRY IS TOO LONG!'
   END IF
END
```

WRITE Statement

The WRITE statement outputs data to a specified unit.

```
WRITE ( [UNIT=] unit [ [, write-spec ]... ] )
  [ output-item [, output-item ]... ]
```

unit

is one of the following:

- An integer expression from 1 through 999 that corresponds to a unit previously connected by an OPEN statement or to a preconnected unit.
- An asterisk (*), which implies unit 6. Unit 6 is preconnected for formatted sequential output.
- An internal file identifier.

If you omit the UNIT keyword, *unit* must be the first item in the list.

For information about external and internal files, see External and Internal Files on page 5-3.

write-spec

is one of the following:

[FMT] = *format*

format is either the label of a FORMAT statement in the same program unit, the name of an integer variable that has been assigned the label of a FORMAT statement, the name of a character array that contains a format specification, a character expression that yields a format specification, or an asterisk (*) indicating a list-directed WRITE statement.

If you omit the optional FMT keyword, *format* must be the second item on the control list and *unit*, without the UNIT specifier, must be the first item on the list.

REC = *recno*

recno is an INTEGER*2 or INTEGER*4 expression that specifies the record number of the record to write.

IOSTAT = *ios*

ios is a variable or array element of integer type that returns an error number or zero (no error) following the WRITE operation. For more information about error numbers, see the Error Numbers on page 6-5.

`ERR = lbl`

lbl is the label of an executable statement in the same program unit to which control passes if an I/O error occurs during a WRITE statement.

`UNLOCK = unlock`

unlock is a logical expression that indicates how the file is to be shared among two or more processes.

`UPDATE = upd`

upd is a logical expression that indicates whether records are to be updated, added, or deleted.

`LENGTH = len`

len is an INTEGER*2 variable or array that helps you determine the last byte position explicitly set when writing to an internal file.

`TIMEOUT = to`

to is an INTEGER*2 or INTEGER*4 expression that specifies (in hundredths of a second) the maximum amount of time to wait for the WRITE operation to complete.

`MSGNUM = msgno`

msgno is an integer expression. For additional information, see Section 14, Interprocess Communication.

`REPLY = reply`

reply is an integer expression. For additional information, see Section 14, Interprocess Communication.

output-item

is a valid expression, array name, array element name, RECORD name, or implied DO list.

Considerations

- Order of control specifiers

  You can write the control specifiers in any order except that if you omit the UNIT keyword, *unit* must be the first item in the list. If you omit the FMT keyword, you must specify *format* as the second item in the list, and the first item must be the unit specifier without the UNIT keyword.

- Use of WRITE statement

Execution of a WRITE statement causes values to be transferred from the output list to the specified file.

The first character of each record has the effect defined by the SPACECONTROL specifier in the OPEN statement.

Execution of a WRITE statement for a file that does not exist causes that file to be created. This does not apply to structured files.

- Use of WRITE with EDIT format files

  Use the INQUIRE statement's NEXTREC specifier to obtain the line number of the last line written.

  If the WRITE statement writes more than one line, the system forms each successive record number by adding 1000 to the last record number.

  If the file was not opened with ACCESS = 'DIRECT' and the WRITE statement does not specify *upd* = .TRUE., the WRITE statement deletes all lines (if any) from the current position to the end of the file before writing the new line or lines.

  You cannot WRITE to an EDIT format file in a FORTRAN program that has been compiled to run as a NonStop process. A program compiled with ENV OLD in effect is compiled to run as a NonStop process if it includes either a CHECKPOINT or START BACKUP statement. A program compiled with ENV COMMON in effect is compiled to run as a NonStop process if it includes a NONSTOP directive.

- REC specifier

  The system interprets negative record numbers according to the file type, as follows:

File Type	REC = -1	REC = -2
EDIT format	Beginning of file	End of file
Other unstructured	End of file	End of file
Relative	End of file	First unused record position

- UNLOCK specifier

  Use the UNLOCK specifier to coordinate file access when a file is shared among two or more processes.

  If the value of *unlock* is .TRUE., FORTRAN uses the WRITEUPDATEUNLOCK procedure to restore access to the current record for the other processes that share the file. For additional information about file and record locking, see the *ENSCRIBE Programmer's Guide*.

- UPDATE specifier

  The UPDATE specifier enables you to update, add, or delete records in structured files.

If the value of *upd* is .TRUE., FORTRAN uses the WRITEUPDATE procedure to update or delete a record. If the UNLOCK specifier is also present, FORTRAN uses the WRITEUPDATEUNLOCK procedure.

If you omit this specifier or specify .FALSE. for *upd*, FORTRAN uses the WRITE procedure to add a record to the file. For additional information about these procedures, see the *ENSCRIBE Programmer's Guide*.

- LENGTH specifier

  The LENGTH specifier returns the last byte position explicitly set when writing to an internal file. When writing to an internal file that can have multiple records, *len* must be an array with at least as many elements as there are records in the internal file.

  Following execution of the WRITE statement, each element of *len* that corresponds to a record not reached by the WRITE operation contains a value of -1; each element that corresponds to a record that was written contains the number of bytes explicitly set.

- TIMEOUT specifier

  If the WRITE operation has not completed by the time *to* (where *to* is the timeout value) centiseconds have elapsed, statement execution terminates with file system error 40; if you use the IOSTAT specifier, *ios* returns 40.

  You can use the TIMEOUT specifier to impose a time limit only if you specify TIMED=.TRUE. when you open the file.

- MSGNUM and REPLY specifiers

  For information about these specifiers, see Section 14, Interprocess Communication.

- Implied DO list

  See Using Implied DO Lists on page 5-27.

- Writing to unit 6

  If you write a record to unit 6 and you have not already established a connection for the unit, WRITE implicitly opens the unit using default parameters. If you specify ENV COMMON and you write a record to unit 6, your FORTRAN routines share access to standard output with routines written in other languages only if the access mode for the unit is OUTPUT. However, the default access mode for unit 6 is I-O. If you want to share access to the file connected to unit 6, you must set the unit's access mode to OUTPUT before you execute the WRITE statement. You can set the access mode:

  In a FORTRAN OPEN statement, as in

```
OPEN( 6, MODE = 'OUTPUT')
```

In a TACL ASSIGN command, as in

```
ASSIGN FT006, , OUTPUT
```

In a UNIT compiler directive, as in

```
UNIT (6, OUTPUT)
```

A PRINT statement also opens unit 6 implicitly.

For more information about using unit 6 as a shared file, see the OPEN Statement on page 7-70.

- If a WRITE statement causes unit 6 to be implicitly opened and your program is running as a NonStop process, the FORTRAN run-time library does a stack checkpoint to the backup process as a part of the implicit open.
- Error conditions

  If you specify *lbl*, and an error occurs during the write operation, the WRITE statement terminates, the file position becomes indeterminate, and FORTRAN transfers control to the statement identified by lbl. If you also specify *ios*, you can determine the error that occurred by analyzing *ios*.

  If you specify *ios*, but not *lbl*, and an error occurs during the write operation, your program continues executing with the statement that follows the WRITE statement. You can analyze ios to determine the error that occurred, if any.

  If you do not specify *ios* or *lbl*, and an error occurs, FORTRAN terminates your program and displays a run-time diagnostic message.

Examples

```
WRITE(UNIT=6) ((array(j,k), k=1,2), J=1,2)
WRITE(payfile, ERR=20, UPDATE=.TRUE.,UNLOCK=.TRUE.) payrec
WRITE(4,200, ERR=250) name, address1, address2, codenumber
```

8 Intrinsic Functions

Intrinsic functions are compiler-defined procedures that return a single value. Topics covered in this section include:

Table 8-1 on page 8-2 summarizes the FORTRAN intrinsic functions described in this section.

The general form of an intrinsic function reference is:

```
function-name ( argument-list )
```

function-name is the generic or specific name of a FORTRAN intrinsic function.

argument-list consists of one or more expressions separated by commas. Each argument you supply must correspond to the type required by function-name.

Declaring Intrinsic Functions

FORTRAN defines the type returned by each intrinsic function. If your program includes a type declaration that specifies the name of an intrinsic function, the intrinsic function's name retains its meaning only if the type you specify is the same as the type of the intrinsic defined in the FORTRAN environment.

If your type declaration specifies a type other than the intrinsic function's type, or if you use the name of an intrinsic function as the name of a variable, array, RECORD, dummy argument, subprogram, or statement function in your program, the intrinsic function's name takes on the meaning you define. Within that program unit, you cannot reference the intrinsic.

If you refer to an external function with the same name as an intrinsic function name, you must declare the external function in an EXTERNAL statement. For additional information, see Section 7, Statements.

If you pass an intrinsic function name as an argument in a CALL statement or external function reference, and that intrinsic function name has not appeared as a referenced function anywhere in the same program unit, FORTRAN assumes the name is a variable rather than an intrinsic function, unless you declare the name as the name of an intrinsic function in an INTRINSIC statement described in Section 7, Statements.

Referencing an Intrinsic Function

You can reference an intrinsic function in a main program or in a subprogram.

You reference an intrinsic function by using it in an expression. For example, in the following assignment statement, FORTRAN evaluates the MAX function to determine the largest value in its argument list, multiplies the result of the MAX function by 5, and stores the result in X:

```
x = MAX (dmon, tue, wed, thu, fri) * 5
```

Table 8-1. FORTRAN Intrinsic Functions (page 1 of 2)

Generic Function Name	Description of Returned Value
ABS	Absolute value
ACOS	Arccosine expressed in radians
AIMAG	Imaginary part of a complex number
AINT	Integer after truncation
ANINT	Nearest whole number
ASIN	Arcsine expressed in radians
ATAN	Arctangent expressed in radians
ATAN2	Arctangent expressed in radians
CHAR	Character value of a specified position in the ASCII collating sequence
CMPLX	Conversion of any numeric type to complex
CONJG	Conjugate of a complex number
COS	Cosine of an angle in radians
COSH	Hyperbolic cosine
DBLE	Conversion of any numeric type to double precision
DIM	Positive difference
DPROD	Double precision product
EXP	Exponential
FILENUM	Guardian file number of a connected unit
ICHAR	Character position of a character in the ASCII collating sequence
INDEX	Starting position of a substring within a string
INT	Conversion of any numeric type to integer
LEN	Length of a character string
LOG	Natural logarithm
LOG10	Base 10 logarithm
MAX	Largest value

Table 8-1. FORTRAN Intrinsic Functions (page 2 of 2)

Generic Function Name	Description of Returned Value
MIN	Smallest value
MOD	Remainder
NINT	Nearest integer
REAL	Conversion of any numeric type to real
SIGN	Value after transferring a sign
SIN	Sine of an angle in radians
SINH	Hyperbolic sine
SQRT	Square root
TAN	Tangent of an angle in radians
TANH	Hyperbolic tangent

Using Generic and Specific Function Names

When you reference an intrinsic function using its generic name, the function accepts arguments of several types, and returns a value whose type corresponds to the type of the arguments you supply when you invoke the function. For example, if you supply a list of real values in a MAX function, MAX returns a real value; if you supply a double precision argument in an EXP function, EXP returns a double precision value.

The generic functions which perform type conversions (CMPLX, DBLE, INT, and REAL) are the only exception to this rule.

When you reference an intrinsic function using its specific name, for example DSIN, the function accepts arguments of a predetermined type. For example, the function AMAX0 selects the largest integer from its argument list and returns it as a real value.

Generic names simplify references to intrinsic functions. The generic and specific names of some intrinsic functions are identical. However, because the compiler replaces the generic name with the specific name that corresponds to the type of the function's arguments, your program always invokes the correct intrinsic function.

Because HP FORTRAN includes integer types of different sizes—INTEGER*2, INTEGER*4, and INTEGER*8—in addition to the standard INTEGER type, all intrinsic functions that return INTEGER according to the ANSI standard, return INTEGER*2 in HP FORTRAN. HP FORTRAN provides analogous intrinsic functions for types INTEGER*4 and INTEGER*8.

This section describes FORTRAN intrinsic functions. The functions are listed in alphabetical order according to their generic type. For example, all sine routines, including ASIN, DSIN, and so forth, are listed under the sine function.

In each function description, the generic function name is listed first in the table of acceptable function names. Argument and function types are not specified for the generic entry because they can be any valid type for the function. The valid types for

the arguments and return value are shown in the entries that follow the generic name in the table.

Considerations

- All items in the argument list of an intrinsic function must be the same type, unless the function's description specifies otherwise.
- When you use an intrinsic name as an actual argument, you must use a specific function name, not a generic function name. In other contexts, you can use either a generic or specific function name.
- You can use either a specific or a generic function name in an INTRINSIC statement.
- If you use a specific or generic function name as a dummy argument, FORTRAN does not interpret it as an intrinsic function reference within that program unit.

ABS Function

The ABS function returns the absolute value of its argument.

The following table describes the argument and function type of the generic ABS function and its associated specific functions.

Syntax	Argument Type	Function Type
ABS (*x*)		
ABS (*x*)	Real	Real
IABS (*x*)	Integer	Integer
IABS4 (*x*)	Integer*4	Integer*4
IABS8 (*x*)	Integer*8	Integer*8
DABS (*x*)	Double Precision	Double Precision
CABS (*x*)	Real	Real

x is an arithmetic expression.

Considerations

- For an integer, real, or double precision argument, the result type of the ABS function is the same as the type of its argument.
- For a complex argument z = CMPLX (*x*, *y*), the value of ABS (*z*) is defined as

$$\sqrt{X^2 + Y^2}$$

and its type is real.

Examples of the ABS Function

The following program prints a value of 530.01 for X, 530.01 for Y, and 0.5E-03 for Z:

```
COMPLEX xnumber
xnumber = (5.3e2, -3.2)
realnum = -.0005
x = CABS (xnumber)
y = ABS (xnumber)
z = ABS (realnum)
PRINT '(2F9.2, E9.1)', x, y, z
END
```

ACOS Function

The ACOS function returns an arccosine expressed in radians.

The table below shows the argument and function type for the generic ACOS function and its associated specific function.

Syntax	Argument Type	Function Type
ACOS (*x*)		
ACOS (*x*)	Real	Real
DACOS (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the ACOS function is the same as the type of its argument.
- The value of ACOS (*x*), where ABS (*x*) ≤1.0, is the angle a (in radians) such that

  $0 \leq a \leq \pi$ and *x* = COS (a)
- If ABS (*x*) > 1.0, program execution terminates abnormally with an error message.

Example of the ACOS Function

FORTRAN stores the value .967937 in Y:

```
y = ACOS (.567)
```

AIMAG Function

The AIMAG function returns the imaginary part of a complex number.

```
AIMAG ( z)
```

z

is a complex expression

Considerations

The AIMAG function returns the imaginary part of a complex number. Its type is always real. That is, if

```
z = CMPLX (x, y)
```

then,

```
AIMAG (z) = y
```

Example of the AIMAG Function

In the following example, if the value (4E3,-2.1) is read into XNUMBER, the value -2.1 is stored in Y.

```
COMPLEX xnumber
READ (*, *) xnumber
y = AIMAG (xnumber)
```

AINT Function

The AINT function returns the non-fractional part of a number.

The table below shows the argument and function type of the AINT generic function and of its associated specific function.

Syntax	Argument Type	Function Type
AINT (*x*)		
AINT (*x*)	Real	Real
DINT (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the AINT function is the same as the type of its argument.
- If the absolute value of *x* is less than 1.0, AINT returns zero.

- If ABS (*x*) > 1.0, AINT returns the integer whose magnitude is the largest integer that does not exceed the magnitude of x and whose sign is the same as that of *x*.

Examples of the AINT Function

```
x = DINT (y)
a = AINT (.01 * c)
```

ANINT Function

The ANINT function returns the whole number that is closest in value to its argument.

The table below shows the argument and function type for the ANINT function and its associated specific function.

Syntax	Argument Type	Function Type
ANINT (*x*)		
ANINT (*x*)	Real	Real
DNINT (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the ANINT function is the same as the type of its argument.
- If x is zero, ANINT returns zero.
- If x is greater than zero, ANINT returns:

  ```
  AINT (x + 0.5)
  ```

- If x is less than zero, ANINT returns:

  ```
  AINT (x - 0.5)
  ```

Example of the ANINT Function

```
esttax = ANINT (price) * .07 * items
```

ASIN Function

The ASIN function returns an arcsine expressed in radians.

The table below shows the argument and function type for the generic ASIN function and its associated specific function.

Syntax	Argument Type	Function Type
ASIN (*x*)		
ASIN (*x*)	Real	Real
DASIN (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the ASIN function is the same as the type of its argument.
- The value of ASIN (*x*), where ABS (*x*) < 1.0, is the angle a (in radians) such that

  $-\pi/2 \leq a \leq \pi/2$ and *x* = SIN (a)
- If ABS (*x*) > 1.0, program execution terminates abnormally with an error message.

Example of the ASIN Function

The value .602859 is stored in Y:

```
y = ASIN (.567)
```

ATAN Function

The ATAN function returns an arctangent expressed in radians.

The table below shows the argument and function type for the ATAN generic function and its associated specific function.

Syntax	Argument Type	Function Type
ATAN (*x*)		
ATAN (*x*)	Real	Real
DATAN (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the ATAN function is the same as the type of its argument.
- The value of ATAN (*x*), for any *x*, is the angle a (in radians) such that

  $-\pi/2 \leq a \leq \pi/2$ and x = TAN (*a*)

Example of the ATAN Function

```
v = ATAN (x)
```

ATAN2 Function

The ATAN2 function returns an arctangent expressed in radians.

The table below shows the argument and function type for the ATAN2 generic function and its associated specific function.

Syntax	Argument Type	Function Type
ATAN2 (*y, x*)		
ATAN2 (*y, x*)	Real	Real
DATAN2 (*y, x*)	Double Precision	Double Precision

y and *x* are arithmetic expressions.

Considerations

- The result type of the ATAN2 function is the same as the type of its arguments. Both arguments must be the same type.
- The value of ATAN2 (*y, x*) is the angle a (in radians) such that

  $-\pi < a \leq \pi$ and y/x = TAN (a)

  and the quadrant of a is determined by the signs of the ordinate y and the abscissa x as follows:

  if y < 0 then *a* < *0*

  if x < 0 then |a| > π/2

  if x = 0 then |a| = π/2

- If x and y are both zero, program execution terminates abnormally with an error message.

Example of the ATAN2 Function

```
a = ATAN2 (4.0, 5.0)
```

CHAR Function

The CHAR function returns the character value of a specified position in the ASCII collating sequence.

```
CHAR ( n )
```

n

is an integer expression with a value from 0 through 127.

Considerations

- The CHAR function result is always type character*1.
- The value of CHAR (*n*) is undefined for argument values of

  $n \leq 0$ or $n > 127$
- The CHAR function is the inverse of the ICHAR function.
- The ASCII collating sequence is given in Appendix A, ASCII Character Set.

Examples of the CHAR Function

The following example returns a value of “Z” for LETTER and a value of “#” for SIGN:

```
CHARACTER letter, sign
letter = CHAR (90)
sign = CHAR (35)
```

CMPLX Function

The CMPLX function returns the complex value of any numeric type.

```
CMPLX (x)
      (x, y)
```

x

is an expression of integer, real, double precision, or complex type. See Considerations.

y

is an expression of integer, real, or double precision type.

Considerations

- If you specify two arguments, they must be the same type: integer, real, or double precision. The data type of the value returned by CMPLX is always complex.
- For one argument of complex type, CMPLX (*x*) returns x. For one argument of integer, real, or double precision type, CMPLX (*x*) is equivalent to CMPLX (*x*, 0).
- If you specify both arguments, CMPLX (*x, y*) returns a complex number with a real part equal to REAL (*x*) and an imaginary part equal to REAL (*y*).

Example of the CMPLX Function

```
REAL a
COMPLEX x
READ (*, *) a
x = CMPLX (a)
```

CONJG Function

The CONJG function returns the conjugate of a complex number.

```
CONJG ( z )
```

z

is an arithmetic expression of complex type.

Considerations

- If the argument is z = CMPLX (*x*, *y*), then the value of CONJG (*z*) is defined as

  CMPLX (*x*, - *y*)

- Note that for any complex number *z* = CMPLX (*x*, *y*),

  CABS (*z*) = $\sqrt{REAL(Z*CONJG(Z))}$

  and

  z * CONJG (*z*) = CMPLX (x^2 + y^2, 0)

Example of the CONJG Function

```
COMPLEX a, b
b = CONJG (a)
```

COS Function

The COS function returns the cosine of an angle expressed in radians.

The table below shows the argument and function type for the COS generic function and its associated specific functions.

Syntax	Argument Type	Function Type
COS (*x*)		
COS (*x*)	Real	Real
DCOS (*x*)	Double Precision	Double Precision
CCOS (*x*)	Complex	Complex

x is an arithmetic expression.

Considerations

- The result type of the COS function is the same as the type of its argument.
- For a real or double precision argument, the function value is always in the range

```
-1.0 ≤COS ( x) ≤1.0
```

- For a complex argument z = CMPLX (*x, y*), the value of COS (*z*) is defined as

```
CMPLX (COS (x) * COSH ( y), -SIN (x) * SINH (y))
```

Example of the COS Function

```
x = COS (.0572)
```

COSH Function

The COSH function returns a hyperbolic cosine.

The table below shows the argument and function type for the COSH generic function and its associated specific function.

Syntax	Argument Type	Function Type
COSH (*x*)		
COSH (*x*)	Real	Real
DCOSH (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the COSH function is the same as the type of its argument.

  The value of COSH (*x*) is defined as

  ```
  (EXP (x) - EXP (- x))/2.0
  ```

- Note that for all values of x

  ```
  COSH (- x) = COSH (x)
  ```

  and

  ```
  COSH (x) ≥ 1.0
  ```

Example of the COSH Function

```
y = COSH (x)
```

DBLE Function

The DBLE function returns a double precision value.

```
DBLE ( x)
```

x

is an expression of any numeric type.

Considerations

- For an argument of double precision type, DBLE (*x*) returns *x*.
- For an argument of integer or real type, DBLE (*x*) returns a value with as much precision of the significant part of x as a double precision datum can contain.
- For an argument of complex type, DBLE (*x*) returns the real part of x with as much precision of the significant part of x as a double precision datum can contain.

Example of the DBLE Function

```
DOUBLE PRECISION tincome
tincome = DBLE (pay) * population
```

DIM Function

The DIM function returns a positive difference.

The table below shows the argument and function type for the DIM generic function and its associated specific functions.

Syntax	Argument Type	Function Type
DIM (*x, y*)		
DIM (*x, y*)	Real	Real
IDIM (*x, y*)	Integer	Integer
IDIM4 (*x, y*)	Integer*4	Integer*4
IDIM8 (*x, y*)	Integer*8	Integer*8
DDIM (*x, y*)	Double Precision	Double Precision

x and *y* are arithmetic expressions.

Considerations

- The result type of the DIM function is the same as the type of its arguments. Both arguments must be the same type.
- The value of DIM (*x, y*) is

  x - y, if *x* > *y*

  y - x, if *x* ≤ *y*.

Example of the DIM Function

```
difference = DIM (a, b)
```

DPROD Function

The DPROD function returns a double precision product.

```
DPROD (x, y)
```

x

y

are arithmetic expressions of type real.

Considerations

- Both arguments to DPROD are type real, but the DPROD function result is type double precision.
- The value of DPROD (*x, y*) is defined as:

```
DBLE (x) * DBLE (y)
```

Example of the DPROD Function

```
      REAL cost (100)
      DOUBLE PRECISION ttax, tax
      ttax = 0
      DO 20 j = 1, 100
         tax = DPROD (cost (j), .07)
         ttax = ttax + tax
20 CONTINUE
```

EXP Function

The EXP function returns an exponential.

The table below shows the argument and function type for the EXP generic function and its associated specific functions.

Syntax	Argument Type	Function Type
EXP (*x*)		
EXP (*x*)	Real	Real
DEXP (*x*)	Double Precision	Double Precision
CEXP (*x*)	Complex	Complex

x is an arithmetic expression.

Considerations

- The result type of the EXP function is the same as the type of its argument.
- For a real or double precision argument x, the value of EXP(*x*) is

  e^x

  where e is the natural logarithm base, or approximately

  2.71828 18284 59045

- For a complex argument z = CMPLX*(x, y*), the value of EXP(*z*) is defined as:

```
CMPLX (EXP(x) * COS(y), EXP(x) * SIN(y))
```

Example of the EXP Function

If the value 8 is stored in X, the following statement stores the value 2980.96 in Y:

```
y = EXP (x)
```

FILENUM Function

The FILENUM function returns the Guardian open file number of the file associated with the specified unit number.

```
FILENUM (k)
```

k

is an integer expression ranging from 1 through 999 specifying the unit number to which the file is connected.

Considerations

- The value of FILENUM (*k*) is defined as:

  -2 if unit number *k* is not defined

  -1 if unit number *k* is defined but is not open

  ≥0 if unit number *k* is open

- The function value can be used as the file number in calls to Guardian procedures that require a file number parameter.

Example of the FILENUM Function

The following example uses the FILENUM function to disable the echo on a terminal:

```
?GUARDIAN SETMODE
      CHARACTER * 10 s
      OPEN (UNIT = 4)
   10 CONTINUE
C Disable echo and do a READ with PROMPT = >:
      CALL setmode (FILENUM (4), 20, 0)
      READ (UNIT = 4, FMT = 1000, PROMPT= '>', END = 20) s
1000 FORMAT (A10)
      .
C Leave the terminal with echo enabled!!!
   20 CALL setmode (FILENUM (4), 20, 1)
      STOP
      END
```

ICHAR Function

The ICHAR function converts a character datum to an integer which represents the character's position in the ASCII collating sequence, as shown in Appendix A, ASCII Character Set.

The table below shows the argument and function type for the ICHAR generic function and its associated specific functions.

Syntax	Argument Type	Function Type
ICHAR (*c*)	Character*1	Integer
ICHAR4 (*c*)	Character*1	Integer*4
ICHAR8 (*c*)	Character*1	Integer*8

c is a character expression with a length of one.

Considerations

- The result of the ICHAR function is always in the range

  $0 \leq \text{ICHAR}(c) \leq 255$

- The ICHAR function is the inverse of the CHAR function.

Example of the ICHAR Function

The following program stores the number 35 in I and J.

```
CHARACTER name
name = '#'
i = ICHAR (name)
j = ICHAR ('#')
PRINT *, i, j
END
```

INDEX Function

The INDEX function returns an integer that points to the first character of a substring relative to the string that contains it.

The table below shows the argument and function type for the INDEX generic function and its associated specific functions.

Syntax	Argument Type	Function Type
INDEX (*string, sub*)	Character	Integer
INDEX4 (*string, sub*)	Character	Integer*4
INDEX8 (*string, sub*)	Character	Integer*8

string

is a character expression whose value is to be searched.

sub

is a character expression whose value is the substring to be found within *string*.

Considerations

- INDEX returns the starting position of the first occurrence of *sub* in *string*.
- INDEX returns a value of 0 if *sub* does not occur in *string* or if the length of *sub* exceeds that of *string*.
- Blank positions in character string values are significant. For example,

```
CHARACTER * 15 password, try
password = 'impassable'
try = 'able'
n = INDEX (password, try)
```

sets N = 0, because the two variables are matched as follows:

```
password = 'impassable^^^^^'
try = 'able^^^^^^^^^^'
```

Example of the INDEX Function

The following example returns a value of 1 for POSITION:

```
RECORD title
CHARACTER * 10 author
CHARACTER * 10 name
END RECORD
CHARACTER * 5 sub
sub = 'Ander'
title^author = 'Anderson'
position = INDEX (title^author, sub)
PRINT *, position
```

INT Function

The INT function returns a value of integer type.

The table below shows the argument and function type of the generic INT function and its associated specific functions:

Syntax	Argument Type	Function Type
INT (*x*)	Arithmetic*	Integer
INT4 (*x*)	Arithmetic*	Integer*4
INT8 (*x*)	Arithmetic*	Integer*8
IFIX (*x*)	Real	Integer
IFIX4 (*x*)	Real	Integer*4
IFIX8 (*x*)	Real	Integer*8
IDINT (*x*)	Double Precision	Integer
IDINT4 (*x*)	Double Precision	Integer*4
IDINT8 (*x*)	Double Precision	Integer*8

x is an arithmetic expression.

* Arithmetic means the argument can be an integer, real, double precision, or complex number.

Considerations

- For an argument of integer type, INT (*x*) returns *x*.
- For an argument of real or double precision type:

if | *x*| < 1.0	INT (*x*) returns 0.
if | *x*| ≥ 1.0	INT (*x*) returns the integer whose magnitude is the largest integer that does not exceed the magnitude of *x* and whose sign is the same as that of x.

  For example,

```
INT (-3.8) returns -3.
```

- For an argument of complex type, INT (*x*) is the same as INT (REAL (*x*)).

Example of the INT Function

```
y = INT (x / 3)
p = x + IFIX4 (s * r)
```

LEN Function

The LEN function returns the declared length of a character item.

The table below shows the argument and function type of the generic LEN function and its associated specific functions:

Syntax	Argument Type	Function Type
LEN (*string*)	Character	Integer
LEN4 (*string*)	Character	Integer*4
LEN8 (*string*)	Character	Integer*8

String is a character expression.

Considerations

If the argument to LEN is a character variable, LEN returns the declared length of the variable, not the length of a string that you assigned to the variable. In the following example, LEN returns 20, the length of NAME, not 7, the length of the string assigned to NAME:

```
CHARACTER * 20 name
INTEGER name_len
name = 'Amadeus'
name_len = LEN(name)
```

LEN is particularly useful to determine the length of a character type argument to a subroutine. See the following example.

Example of the LEN Function

```
SUBROUTINE word (entry, length)
CHARACTER entry * (*)
length = LEN (entry)
IF (length .GT. 80) THEN
  PRINT *, 'Your entry is too long!'
  END IF
END
```

LOG Function

The LOG function returns a natural (or base e) logarithm.

The table below shows the argument and function type for the LOG generic function and its associated specific functions.

Syntax	Argument Type	Function Type
LOG (*x*)		
ALOG (*x*)	Real	Real
DLOG (*x*)	Double Precision	Double Precision
CLOG (*x*)	Complex	Complex

x is an arithmetic expression.

Considerations

- The result type of the LOG function is the same as the type of its argument.
- For a real or double precision argument x, the value of LOG (*x*) is the number Y that:

  $x = e^y$

  where e is the natural logarithm base, or approximately

  ```
  2.71828 18284 59045
  ```

  If $x < 0$, program execution terminates abnormally with an error message.
- For a complex argument z = CMPLX (*x*, *y*), the value of LOG(*z*) is defined as

  ```
  CMPLX (ALOG (CABS (z)), ATAN2 (y, x))
  ```

  If $x = 0$ and $y = 0$, program execution terminates abnormally with an error message.

- The value returned by the LOG function in programs that specify ENV OLD might differ slightly from the value returned by the LOG function in programs that specify ENV COMMON. The value returned by the LOG function when you specify ENV COMMON differs slightly because of rounding methods on the final result.

Example of the LOG Function

```
x = ALOG (a) + ALOG (b)
```

LOG10 Function

The LOG10 function returns a common (or base 10) logarithm.

The table below shows the argument and function type for the LOG10 generic function and its associated specific functions.

Syntax	Argument Type	Function Type
LOG10 (*x*)		
ALOG10 (*x*)	Real	Real
DLOG10 (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The value of LOG10 (*x*) is the number y such that

  $x = 10.0^{y}$

- The result type of the LOG10 function is the same as the type of its argument.
- If $x < 0$, program execution terminates abnormally with an error message.
- The value returned by the LOG10 function in programs that specify ENV OLD might differ slightly from the value returned by the LOG10 function in programs that specify ENV COMMON. The value returned by the LOG10 function when you specify ENV COMMON differs slightly because of rounding methods on the final result.

Example of the LOG10 Function

```
DOUBLE PRECISION x, a
x = DLOG10 (a)
```

MAX Function

The MAX function returns the largest value in its argument list.

The table below shows the argument and function type for the MAX generic function and its associated specific functions.

Syntax	Argument Type	Function Type
MAX (*x1, ... , xn*)		
MAX0 (*x1, ... , xn*)	Integer	Integer
MAX04 (*x1, ... , xn*)	Integer*4	Integer*4
MAX08 (*x1, ... , xn*)	Integer*8	Integer*8
AMAX0 (*x1, ... , xn*)	Integer	Real
AMAX04 (*x1, ... , xn*)	Integer*4	Real
AMAX08 (*x1, ... , xn*)	Integer*8	Real
MAX1 (*x1, ... , xn*)	Real	Integer
MAX14 (*x1, ... , xn*)	Real	Integer*4
MAX18 (*x1, ... , xn*)	Real	Integer*8
AMAX1 (*x1, ... , xn*)	Real	Real
DMAX1 (*x1, ... , xn*)	Double Precision	Double Precision

Each *xi* is an arithmetic expression.

Considerations

- All arguments to the MAX function must be the same type.
- The result type of the MAX function is the same as the type of its arguments.
- You can specify up to 63 arguments to the MAX function. If you need to use more than 63 arguments, you can distribute the arguments among several MAX functions and take the MAX of those functions:

```
MAX (MAX (a, b, c, ... , x), MAX (a1, b1, c1, ... , x1))
```

Example of the MAX Function

```
READ *, x, y, z
greatest = MAX (x, y, z)
```

MIN Function

The MIN function returns the smallest number from its argument list.

The table below shows the argument and function type for the MIN generic function and for its associated specific functions.

Syntax	Argument Type	Function Type
MIN (*x1, ... , xn*)		
MIN0 (*x1, ... , xn*)	Integer	Integer
MIN04 (*x1, ... , xn*)	Integer*4	Integer*4
MIN08 (*x1, ... , xn*)	Integer*8	Integer*8
AMIN0 (*x1, ... , xn*)	Integer	Real
AMIN04 (*x1, ... , xn*)	Integer*4	Real
AMIN08 (*x1, ... , xn*)	Integer*8	Real
MIN1 (*x1, ... , xn*)	Real	Integer
MIN14 (*x1, ... , xn*)	Real	Integer*4
MIN18 (*x1, ... , xn*)	Real	Integer*8
AMIN1 (*x1, ... , xn*)	Real	Real
DMIN1 (*x1, ... , xn*)	Double Precision	Double Precision

Each *xi* is an arithmetic expression.

Considerations

- All arguments to the MIN function must be of the same type.
- The result type of the MIN function is the same as the type of its arguments.
- You can specify up to 63 arguments to the MIN function. If you need to use more than 63 arguments, you can distribute the arguments among several MIN functions and then take the MIN of those functions:

```
MIN (MIN (a, b, c, ... , x), MIN (a1, b1, c1, ... , x1))
```

Example of the MIN Function

```
REAL mon
day = MIN (mon, tues, wed, thu, fri)
```

MOD Function

The MOD function returns the remainder of x divided by y.

The table below shows the argument and function type for the MOD generic function and its associated specific functions.

Syntax	Argument Type	Function Type
MOD (*x, y*)		
MOD (*x, y*)	Integer	Integer
MOD4 (*x, y*)	Integer*4	Integer*4
MOD8 (*x, y*)	Integer*8	Integer*8
AMOD (*x, y*)	Real	Real
DMOD (*x, y*)	Double Precision	Double Precision

x and *y* are arithmetic expressions.

Considerations

- The types of both arguments to MOD must be the same.
- The value of MOD (*x*, *y*) is defined as

```
x - (INT(x/y) * y)
```

- The result type of the MOD function is the same as the type of its arguments.
- If y is zero, the function is undefined.

Example of the MOD Function

The program below uses MOD to print a blank line every fifth line:

```
      DO 20, j = 1,100
        WRITE (*, 22) customer (k)
        IF (MOD (j, 5) .EQ. 0) THEN
          PRINT *, ' '
          END IF
  20 CONTINUE
```

NINT Function

The NINT function returns the integer that is closest to the value of its argument.

The table below shows the argument and function type of the NINT generic function and its associated specific functions:

Syntax	Argument Type	Function Type
NINT (*x*)	Real	Integer
NINT4 (*x*)	Real	Integer*4
NINT8 (*x*)	Real	Integer*8
IDNINT (*x*)	Double Precision	Integer
IDNINT4 (*x*)	Double Precision	Integer*4
IDNINT8 (*x*)	Double Precision	Integer*8

x is an arithmetic expression.

Considerations

- If *x* is zero, NINT returns zero.
- If *x* is greater than zero, NINT returns:

```
INT ( x + 0.5)
```

- If *x* is less than zero, NINT returns:

```
INT ( x - 0.5)
```

Example of the NINT Function

```
DOUBLE PRECISION tonnage
INTEGER * 8 volume
volume = IDNINT8 (tonnage)
```

REAL Function

The REAL function returns a real type value.

The table below describes the argument and function type of the generic REAL function and of its associated specific functions.

Syntax	Argument Type	Function Type
REAL (*x*)	Integer	Real
FLOAT (*x*)	Integer	Real
SNGL (*x*)	Double Precision	Real

x is an arithmetic expression.

Considerations

- For an argument of real type, REAL (*x*) returns *x*.
- For an argument of integer or double precision type, REAL (*x*) has as much precision of the significant part of *x* as a real datum can contain.
- For an argument of complex type, REAL (*x*) returns the value of the real part of *x*.

  That is,

  ```
  REAL(CMPLX(x, y))
  ```

  returns *X*.
- For an integer argument, FLOAT (*x*) returns the same value as REAL (*x*).

Example of the REAL Function

```
total = FLOAT (number) * 2.50
```

SIGN Function

The SIGN function returns a value after transferring a sign.

The table below shows the argument and function type for the SIGN generic function and its associated specific functions.

Syntax	Argument Type	Function Type
SIGN (*x, y*)		
SIGN (*x, y*)	Real	Real
ISIGN (*x, y*)	Integer	Integer
ISIGN4 (*x, y*)	Integer*4	Integer*4
ISIGN8 (*x, y*)	Integer*8	Integer*8
DSIGN (*x, y*)	Double Precision	Double Precision

x and *y* are arithmetic expressions.

Considerations

- The result of the SIGN function is of the same type as the arguments. Both arguments must be of the same type.
- The value of SIGN (*x, y*) is defined as:

  |x| if $y \geq 0$

  -|x| if $y < 0$

Example of the SIGN Function

```
z = SIGN (a, b)
```

SIN Function

The SIN function returns the sine of an angle expressed in radians.

The table below shows the argument and function type for the SIN generic function and its associated specific functions.

Syntax	Argument Type	Function Type
SIN (*x*)		
SIN (*x*)	Real	Real
DSIN (*x*)	Double Precision	Double Precision
CSIN (*x*)	Complex	Complex

x is an arithmetic expression.

Considerations

- The result type of the SIN function is the same as the type of its argument.
- For a real or double precision argument, the function value is always in the range

  $-1.0 \leq SIN(x) \leq 1.0$

- For a complex argument z = CMPLX (X,Y), the value of SIN(Z) is defined as

```
CMPLX (SIN (x) * COSH (y), COS (x) * SINH (y))
```

Example of the SIN Function

```
x = SIN (y)
```

SINH Function

The SINH function returns a hyperbolic sine.

The table below shows the argument and function type for the SINH generic function and its associated specific function.

Syntax	Argument Type	Function Type
SINH (*x*)		
SINH (*x*)	Real	Real
DSINH (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the SINH function is the same as the type of its argument.
- The value of SINH (*x*) is defined as

  ```
  (EXP (x) - EXP (-x))/2.0
  ```

- Note that

  ```
  SINH (-x) = -SINH (x)
  ```

Example of the SINH Function

```
y = SINH (x)
```

SQRT Function

The SQRT function returns the square root of a number.

The table below shows the argument and function type for the SQRT generic function and its associated specific functions.

Syntax	Argument Type	Function Type
SQRT (*x*)		
SQRT (*x*)	Real	Real
DSQRT (*x*)	Double Precision	Double Precision
CSQRT (*x*)	Complex	Complex

x is an arithmetic expression.

Considerations

- The result type of the SQRT function is the same as the type of its argument.
- For a real or double precision argument $x < 0$, program execution terminates abnormally with an error message.
- For a complex argument *z* = CMPLX (*x*, *y*), the value of SQRT (*z*) is defined as

  CMPLX ($\sqrt{(|Z| + X)/\ 2.0}$, SIGN ($\sqrt{(|Z| - X)/\ 2.0}$, Y°)°)

- If ABS (*z*) < x, program execution terminates abnormally with an error message.

Example of the SQRT Function

```
COMPLEX y, z
z = CSQRT (y)
a = SQRT (b)
```

TAN Function

The TAN function returns the tangent of an angle expressed in radians.

The table below shows the argument and function type for the TAN generic function and its associated specific function.

Syntax	Argument Type	Function Type
TAN (*x*)		
TAN (*x*)	Real	Real
DTAN (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the TAN function is the same as the type of its argument.
- The value of TAN (*x*) is defined as:

  `SIN (`*x*`)/COS (`*x*`)`

  and is undefined for values of *x* for which COS (*x*) = 0.

Example of the TAN Function

```
x = TAN (y)
```

TANH Function

The TANH function returns a hyperbolic tangent.

The table below shows the argument and function type for the TANH generic function and its associated specific function.

Syntax	Argument Type	Function Type
TANH (*x*)		
TANH (*x*)	Real	Real
DTANH (*x*)	Double Precision	Double Precision

x is an arithmetic expression.

Considerations

- The result type of the TANH function is the same as the type of its argument.
- The value of TANH (*x*) is defined as

  `SINH (`*x*`)/COSH (`*x*`)`

Note that

```
TANH (-x) = -TANH (x)
```

Example of the TANH Function

```
y = TANH (x)
```

9 Program Compilation

You run the FORTRAN compiler, which resides in a file named $SYSTEM.SYSTEM.FORTRAN, by using the TACL implied RUN command.

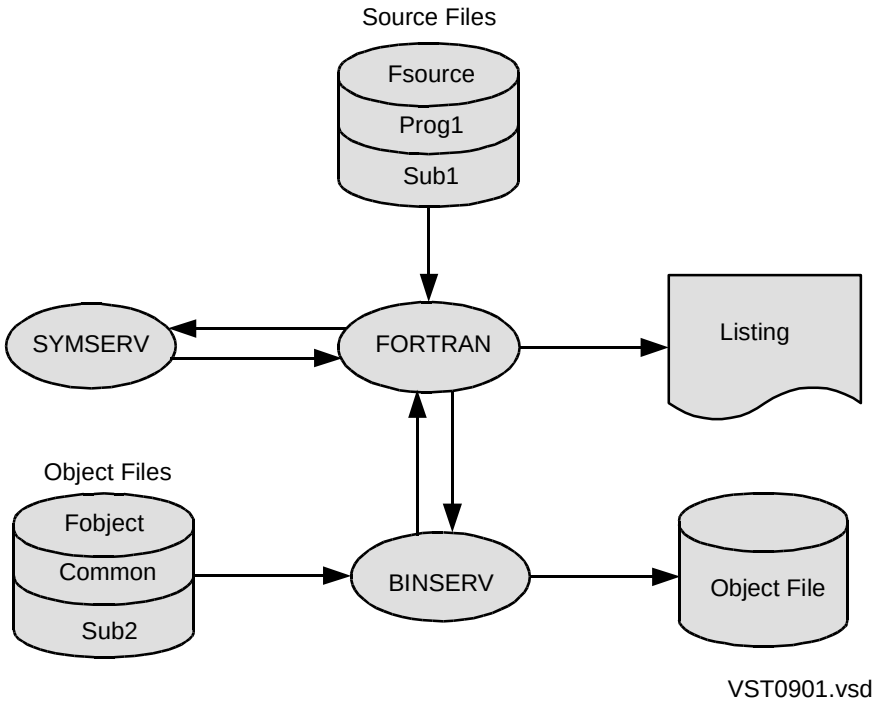
The FORTRAN compiler accepts source files containing FORTRAN statements, comment lines, and compiler directives. Source files alone or in combination with other source files and object files constitute the input to a compilation. Topics covered in this section include:

The compilation of a FORTRAN program involves the following three processes:

- FORTRAN, which compiles code and calls BINSERV and SYMSERV for additional processing as needed. FORTRAN produces all listings after BINSERV and SYMSERV have completed processing.
- BINSERV, the compile-time binder, is present throughout a compilation (unless you specify only syntax checking). BINSERV stops when it detects an error in the source code. Output listings always contain binder statistics if an object file was produced.
- SYMSERV, which produces object-file symbol tables and source level cross-reference information. SYMSERV is present throughout a compilation.

FORTRAN starts BINSERV and SYMSERV automatically. Figure 9-1 shows the relationship among the processes.

Figure 9-1. The Compilation Process



Compiling a Program

The syntax diagram below describes the implied RUN command to compile a FORTRAN program.

```
FORTRAN [/[IN source] [,OUT [ list] ] [, option]... /]
        [ object] [; directive [, directive]...]
```

source

is the name of a disk file, process, $RECEIVE, magnetic tape (unlabeled and unblocked only), class map DEFINE, or terminal from which the compiler reads source-language statements and compiler directives. Disk files can be structured, unstructured, or EDIT format. FORTRAN reads 132-byte records from *source* until it encounters an end-of-file record.

The form of source is one of the following:

[*node*.][$*volume*.][*subvolume*.] *fileid*

$*device-name*

$*logical-device-number*

$*process-name*

If you omit IN *source*, the compiler reads from the TACL IN file, normally the home terminal.

list

is the name of a process (including a spooler collector), class map or spool DEFINE, printer, magnetic tape (unlabeled and unblocked only), terminal, or disk file to which FORTRAN directs its listing output. A disk file can be structured (but not key-sequenced), unstructured, or an EDIT format file. The *list* name uses the same form as *source*. (See Using a Tape or Disk File for the Listing Output on page 9-4 when *list* is a magnetic tape or disk file.) If you omit OUT *list*, FORTRAN uses the TACL OUT file, normally the home terminal. When a disk file name is given that doesn't exist, an EDIT format file is created for the listing output. When *list* is a spooler collector process, the compiler uses Level-3 spooling, regardless of any SPOOLOUT parameter given. If you specify OUT but omit *list*, output is suppressed.

option

is any of the RUN options defined by TACL. The following are among the more frequently used options. For a complete list of run options, see the *TACL Reference Manual*.

CPU *cpu-number*

is an integer ranging from 0 through 15 that specifies in which processor to run the compiler. If you omit this option, the compiler runs in the same processor as TACL. (If your system runs a $CMON process, the $CMON process might assign a different processor for the compilation. For information about $CMON, see the *Guardian Programmer's Guide*.)

PRI *priority*

is an integer ranging from 1 through 199 that specifies the execution priority of the compiler. Processes with higher numbers execute first.

MEM *num-pages*

is an integer ranging from 1 through 64 that specifies the maximum number of virtual data pages to allocate for the process. If you omit this option, FORTRAN allocates 64 pages. If you specify a smaller number, the compiler run usually fails.

NOWAIT

specifies that TACL display a prompt after it sends the startup message to the compiler, rather than waiting for the compilation to complete.

TERM *terminal-name*

is the name of a terminal or process that acts as the home terminal for the compiler. If you omit this option, FORTRAN uses the TACL home terminal.

object

specifies the disk file name that BINSERV gives the compiled object program. If you omit this entry, FORTRAN uses a file named OBJECT on your current system, volume, and subvolume. If OBJECT already exists, BINSERV purges it before creating the target file. If BINSERV cannot name the file with either *object* or OBJECT, it assigns the target file a name in the form of ZZBI *nnnn*, where *nnnn* is a random number.

directive

is a FORTRAN compiler directive described in Section 10, Compiler Directives.

Command Line Length

The FORTRAN command (or any other TACL command) can contain a maximum of 132 characters on the same line. You can enter commands of up to 528 characters by ending each line of the command (except the last) with an ampersand character (&).

The following two commands are equivalent:

```
1> FORTRAN/ IN mortgage, OUT listmort, NOWAIT/;INTEGER*4
1> FORTRAN/ IN mortgage, OUT listmort, NOWAIT/&
2> ;INTEGER*4
```

The directives following the semicolon can have a total length of up to 280 characters.

Examples

The following TACL command compiles the source file BUGS, sends the compiler listing to LIST, and creates the object file OBUGS:

```
1> FORTRAN/ IN bugs, OUT list/ obugs
```

The following TACL command compiles the source file NEWPROG, sends the compiler listing to a printer, and creates the object file OBJECT. It uses the NOWAIT option and the compiler directives INTEGER*4 and ANSI.

```
1> FORTRAN/ IN newprog, OUT $s.#lp, NOWAIT/;INTEGER*4, ANSI
```

Using a Tape or Disk File for the Listing Output

If the device to which FORTRAN writes its listing file is a process, a printer, or a terminal, FORTRAN takes care of vertical format control (skip a line, start a new page, and so on) by issuing CONTROL and SETMODE requests.

When the list file is a magnetic tape or any kind of disk file, the compiler writes each line beginning with a control character (0 to skip a line, 1 to start a new page, and so forth) according to the FORTRAN 77 ANSI standard. You can print such a file correctly by sending it to the spooler with a FORTRAN program such as the following:

```
      INTEGER max rec, rec len
      CHARACTER * 132 line
1 FORMAT (A)
      OPEN (5, ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED',
     X      MODE = 'INPUT', PROTECT = 'PROTECTED')
      OPEN (6, ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED',
     X      MODE = 'OUTPUT', PROTECT = 'EXCLUSIVE')
      INQUIRE (5, RECL = max rec)
10 READ (5, 1, END = 30, LENGTH = rec len) line (1: max rec)
      WRITE (6, 1) line (1: rec len)
      GO TO 10
30 STOP
      END
```

TACL PARAM Commands

Before you execute the FORTRAN compiler, you can enter TACL PARAM commands that alter compile-time attributes of the FORTRAN compiler. Table 9-1 lists the compile-time PARAMs recognized by FORTRAN and shows how they affect your compilation.

Table 9-1. PARAM Commands

PARAM	Effect
OUTWIDTH *n*	Specifies the number of columns to write to your program's listing file.
SAMECPU	Specifies whether to force processes started by the FORTRAN compiler to run in the same CPU as the compiler.
SWAPVOL *volume*	Specifies the name of the volume that the compiler uses for temporary files.

Compiling With FORTRAN and BINSERV in the Same CPU

By default, FORTRAN, BINSERV, and SYMSERV run in different CPUs. If you want to reduce interprocessor message traffic, use the TACL PARAM SAMECPU command to run FORTRAN and BINSERV in the same CPU. The form of the command is

```
1> PARAM SAMECPU n
```

where *n* is a nonzero integer. For example:

```
1> PARAM SAMECPU 3
```

Note that the nonzero value does not designate in which CPU the processes run. Use the TACL CPU run-option when you run your program if you want to specify the CPU in which the processes run.

Specifying a Volume for the Compiler's Temporary Files

Use a TACL PARAM SWAPVOL command to specify a volume other than the logon default volume for the storage of the compilation's temporary files. If you are running the compiler over the network, this command can reduce unnecessary network traffic. In this case, the SWAPVOL volume should be on the same system as the compiler.

The PARAM SWAPVOL command has the form:

```
1> PARAM SWAPVOL [\node.]$volume
```

If you do not use this command, the FORTRAN compiler uses the default volume; BINSERV and SYMSERV use the volume of the object file.

If the compiler cannot create its first temporary file on the specified volume, compilation proceeds as though you did not specify a PARAM SWAPVOL command.

Specifying the Line Length for the Listing File

The OUTWIDTH compiler PARAM specifies the maximum length of the lines written to the compiler's listing output file. The PARAM OUTWIDTH command must precede the FORTRAN command.

The PARAM OUTWIDTH command has the form:

```
1> PARAM OUTWIDTH number
```

where *number* is an unsigned integer in the range 72 through 132. *number* can be less than, equal to, or greater than the file's assumed record length.

The assumed record length of the compiler's OUT file is:

- Its maximum record length if a structured disk file
- Its physical record length if a printer or terminal
- 132 characters if any other kind of file

In the absence of PARAM OUTWIDTH, the compiler writes records of the file's assumed record length.

If PARAM OUTWIDTH number is specified, and

- *number* is at least 72, but less than or equal to the file's assumed record length, the compiler writes records of *number* characters.
- *number* exceeds the file's assumed record length, the compiler formats a line of *number* characters and writes the line as one or more records of the file's assumed record length.
- *number* is less than 72, then the compiler ignores the PARAM entirely. If *number* is greater than 132, the compiler treats it as 132.

For most portions of its listing, the compiler can format line images in two lengths: 80 and 132 characters. If the assumed record length of the compiler's OUT file is at least 132 characters and the PARAM OUTWIDTH specifies at least 132 characters or is not present, the compiler formats output line images of 132 characters. For all other cases, the compiler formats output line images of 80 characters.

Note. It can be helpful to limit the compiler's output to an 80-character width when the listing output file is an EDIT format file that is displayed on a terminal screen, or sent to a spooler location that prints on 8 1/2 by 11 inch paper. Limiting the compiler's output to 80 character lines can also be useful if you are directing the compiler's output to a narrow device such as a terminal, but you want the lines formatted as if for a 132-character wide device.

Compiler Operation

Except when generating certain global tables, the compiler treats each program unit as a separate entity. Global tables include information about procedures and their arguments, I/O units and buffer areas, and the names and lengths (but not the contents) of COMMON data blocks.

The compilation process for a program unit proceeds as follows:

- The compiler parses source code and makes skeletal table entries. Then, it generates intermediate code in the form of "trees," with each tree corresponding to a source statement. The compiler detects any scanning or syntax errors during this stage.
- The compiler completes the symbol table entries. It processes information from COMMON, SAVE, DATA, and EQUIVALENCE statements to make run-time address assignments and to build tables for global processing.
- The compiler emits object code to allocate data storage for local data, and processes the intermediate code to generate the final object code for the program unit.

BINSERV binds the appropriate object modules from FORTLIB (a library of object modules supplied by HP) into the target file. It also binds in code and data blocks located via the SEARCH directive.

Any unresolved references to procedures are represented in the program file in a procedure identifier list. The operating system uses this list to resolve such references the first time you execute the object program. These procedures include any Formatter, I/O procedures, and Guardian procedures you specify in the program.

If you have not bound the procedures you call into a program file prior to execution, and the operating system cannot find them in a run-time library, it displays the following message at run time:

```
UNRESOLVED EXTERNAL
```

You can input the compiled object file to a later compilation or to an interactive Binder session. You can bind other procedures with the target file to create an executable program. Use the SEARCH directive to use the object file as input to a compilation.

For information about binding program units written in C, COBOL85, Pascal, and TAL with FORTRAN program units, see Section 13, Mixed-Language Programming.

Interpreting Compilation Listings

The following is a guide to the interpretation of the various parts of the listing produced by the FORTRAN compiler. The presence or absence of a given listing depends on the listing options you specify. The compiler directives shown in Table 9-2 control listing options (default values are underlined).

For additional information about the listing directives, see Section 10, Compiler Directives.

Table 9-2. Compiler Listing Options (page 1 of 2)

Directive		Action
ANSI	NOANSI	Ignore characters in columns 73 through 132.
CODE	NOCODE	List octal instruction code generated for each program unit.
COLUMNS *n*		Define the line length of each record in a source file.
CROSSREF	NOCROSSREF	Generate cross-reference information for selected identifier classes.
ERRORFILE *file*		Write compilation error messages in *file*.
FMAP	NOFMAP	Include a file map in the listing.
ICODE	NOICODE	List symbolic instruction codes generated for each program unit.
LINES *number*		Write *number* lines to the listing file before skipping page.

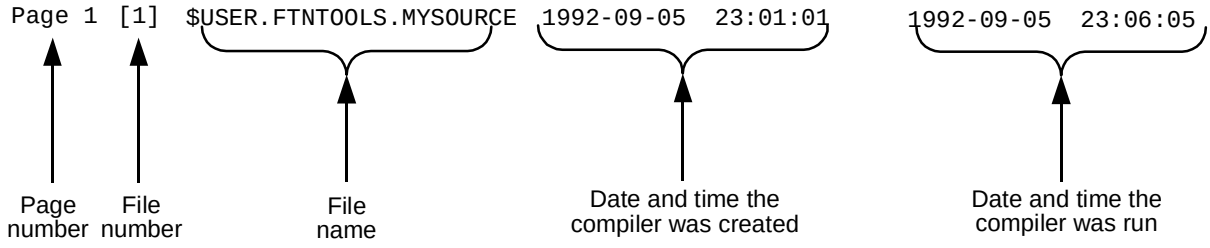
Table 9-2. Compiler Listing Options (page 2 of 2)

Directive		Action
LIST	NOLIST	List source lines: enables CODE, CROSSREF, ICODE, LMAP, MAP, and PAGE.
LMAP	NOLMAP	List load maps.
MAP	NOMAP	List tables of local identifiers for each program unit; list table of entities in common storage.
PAGE *title*		Eject current page of list file (except for first occurrence). Print *title* at the top of next page.
PRINTSYM	NOPRINTSYM	List unreferenced identifiers in map listings.
SUPPRESS	NOSUPPRESS	List only error messages and compilation statistics. Overrides LIST.
WARN	NOWARN	List compiler warning messages.

Page Heading

FORTRAN displays a heading that lists the page number, the file ordinal, file name, and the date and time the source file currently being compiled was last modified, the date and time at the start of compilation, the current program unit type and name, and an optional title (specified by the PAGE directive) at the top of each page of the listing.

Figure 9-2. Compiler Listing—Page Heading



Compiler Heading

The first page of the listing contains a heading that lists the version number and release date of the FORTRAN compiler in use, the default compiler options, and the FORTRAN compiler copyright notice. If you specified other listing options in the RUN

command when you initiated the compilation, FORTRAN lists those directives just before the first line of the source code.

```
Tandem FORTRAN - T9252D10 - (08JUN92)        Default options: on
(LIST,MAP,WARN,LMAP) - off (CODE,ICODE,ANSI)
Copyright Tandem Computers Incorporated 1978, 1979, 1980, 1981, 1982,
1983, 1984, 1985, 1986, 1987, 1988, 1989, 1991
```

Source Listing

If the LIST option is in effect, FORTRAN lists the text of the source program following the compiler heading. Each line of the listing is preceded by a line number that corresponds to its line number in the EDIT format file containing the source code.

The compiler prints information about each line of source text between the line number and the source line image itself. The information is one or two characters, as follows:

Character	Meaning
n	For the first (or only) line of a FORTRAN statement, where n is a one- or two-digit block nesting level. It is zero normally (printed as a blank), and it increases by one for each block-IF or DO-loop that has begun but not yet ended.
-	For each continuation line of a FORTRAN statement.
*	For a FORTRAN comment line (C or c or * in column 1 of the source line, or an allblank source line).
?	For a compiler directive (? in column 1).
#	For an IF-skipped line (a source line that is read and printed but not otherwise processed by the compiler because of skipping initiated by an IF or IFNOT directive and not yet terminated by a matching ENDIF directive).

Example 9-1. Compiler Listing—Source Listing (page 1 of 2)

```
1.  ?  ?ICODE
2.  *  C Program to set up table of lot number, name, property
3.  *  C value and determine tax charge.
4.  *  C
5.  *  C avalue = average property value
6.  *  C atax = average tax
7.  *  C tvalue = sum of property values
8.  *  C ttax = total tax
9.  *  C
10. 40 FORMAT ('0',5x, i4, 5x,a11,5x,f6.0,f7.2)
11. 50 FORMAT ('0',5x,'Lot #',4x, 'Owner''s Name',5x,'Value',4x
12. -  & 'Tax Charge')
13. *  C
14. 100 CHARACTER name*11
15. REAL value, tax, taxch, tvalue, ttax, avalue, atax
16. DATA avalue, atax, ttax, tvalue/4*0/
17. *  C
18. 150 WRITE (6,50)
19. *  C
20. *  C Execute loop to determine taxcharge and to accumulate
21. *  C total value and total taxcharge:
22. *  C
23. 200 DO 400 j = 1,5
24. 1      READ *, lotnum, name, value
25. 1      tvalue = tvalue + value
26. 1        IF (value .LT. 10000) THEN
27. 2           taxchg = .03 * VALUE
28. 2        ELSE
29. 2           taxchg = .04 * VALUE
30. 2        ENDIF
31. 1      WRITE (6,40) lotnum, name, value, taxchg
32. 1      ttax = ttax + taxchg
33. 1  400 CONTINUE
34. *  C
35. 450 atax = ttax/5
36. avalue = tvalue/5
37. *  C
```

Example 9-1. Compiler Listing—Source Listing (page 2 of 2)

```
38. 460 PRINT *, ' '
39. PRINT *, ' '
40. PRINT *, 'Sum of property values is ', tvalue
41. PRINT *, 'Total tax is ', ttax
42. PRINT *, 'Average property values is ', avalue
43. PRINT *, 'Average tax charge is ', atax
44. 500 END
```

The compiler inserts a line that identifies the source input file being read when it begins to read a different source file defined by a SOURCE directive and when it finishes reading the source file.

The format of the inserted print line image is:

```
Source file: [n] file-name yyyy-mm-dd hh:mm:ss
```

where:

[n]

is the file ordinal (same as in the listing page title),

file-name

is the fully qualified source file name

yyyy-mm-dd hh:mm:ss

is the date and time *file-name* was last modified.

The compiler prints error messages and warning messages in the output listing. Note, however, that error messages are associated with FORTRAN statements, not lines. Because of the FORTRAN continuation line convention, it is impossible for the compiler to know that it has seen the last line of a statement until it sees the first line of the next statement: the compiler accumulates the entire statement before it parses the line. Therefore, an error or warning message in the source listing could refer to any item in the preceding statement (including comment lines and blank lines) and is not restricted to the physical preceding line.

If FORTRAN detects a syntax error and you use the NOLIST or SUPPRESS directives to disable listing, FORTRAN displays the text of the last line it read, its EDIT line number, and the ordinal of the file from which the line was read before it writes the error message. However, as noted in the preceding paragraph, the error or warning message refers to the entire last statement, not just to the last line whose text it displays.

If the MAP, CODE, or ICODE option is in effect, the specified list items follow each program unit. The MAP output for each program unit is in two parts: the code and

data blocks, and the symbolic name map. The maps are in addition to the entry point and data block maps that BINSERV provides after all compiler-generated output following the last program unit.

Code and Data Blocks MAP Listing

The MAP option provides a summary of the code and data blocks in the compiled program unit, showing the name of each block the program unit uses and the amount of memory space it occupies. The summary can help you determine each program unit's contribution to the total code size, dynamic data space, and static data space requirements of the executable program. You can use this information, for example, to determine the optimal amounts for the DATAPAGES and LARGESTACK directives when the estimates made by the compiler are not sufficient.Example 9-2 shows the format of this map.

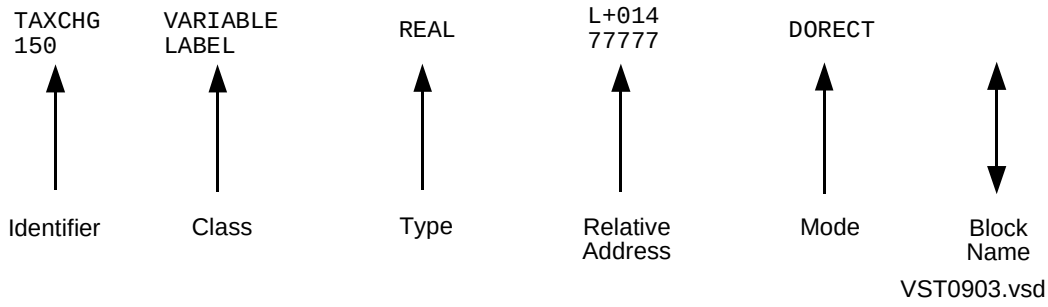
Example 9-2. Compiler Listing—Code and Data Blocks MAP Listing

```
CODE AND DATA BLOCKS   -    Program MAIN^

BLOCK DESCRIPTION              SIZE IN BYTES      BLOCK NAME

Program unit executable code:           1392      MAIN^
Dynamically allocated local data:         40      Standard run-time stack
Statically allocated local data:          16      +MAIN^
Common data blocks:                     None
```

Symbolic Name MAP Listing

The MAP option provides a map of identifiers in alphabetic order. Unreferenced local variables are not listed in the map. FORTRAN does not allocate space for data items that your program does not reference. The map includes the names of unreferenced variables if you specify the PRINTSYM directive. Figure 9-3 shows the format for each line in the map:

Figure 9-3. Compiler Listing—MAP Listing



Each identifier (symbolic name or statement label) in the MAP listing is further qualified by the following information:

Class	Specifies the type of entity represented by the identifier.
Type	Specifies the data type of the identifier. If the identifier does not appear in a data type statement, an exclamation point preceding the entry shows that the data type is implicitly declared.
Relative Address	Is the relative address of the identifier. The relative address is specified as: ● L+ nnn for variables and arrays in local storage. ● L- nnn for dummy arguments of a subprogram. ● LX+ nnn for dynamically allocated local data items in the extended data segment. ● &nnn for a statically allocated data items in the extended data segment. ● C+ nnn for entities in common (where nnn represents an offset from the start of the common block. The actual offset is shown later in the common map). ● nnnnn for statement labels (an offset from the start of the program unit. The actual offset is shown later in the load map). The map does not include an entry for a program unit name.
Mode	States whether the address mode of a variable or array is direct or indirect. Integer, logical, and real variables in local storage are DIRECT. Variables of all other data types, arrays, and entities that appear in COMMON, SAVE, or DATA statements, or that are equivalenced to such entities are INDIRECT.
Block Name	Is the name of the common block where the identifier resides, if it is an entity in common. FORTRAN indicates the unnamed common block as BLANK^. All items having extended addresses are EXTENDED.

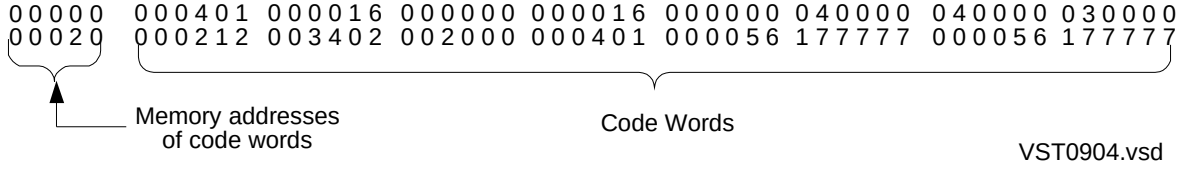
You can use MAP listing information with DEBUG and low-level Inspect which use memory addresses to reference program locations.

The load map lists the address of the common blocks.

CODE Listing

The CODE option lists the instruction codes for a program unit.

Figure 9-4. Compiler Listing—CODE Listing



Each address listed is the octal starting address for the adjacent code, expressed as an offset from the beginning of the program unit. The code is the octal representation of the instruction code emitted by the compiler.

ICODE Listing

The ICODE option provides a listing of the symbolic form of the instruction code for the program unit. The listing contains references to code “landmarks”; for example, START OF PROGRAM UNIT, STATEMENT LABEL 200 + 2, and so forth. Use these to locate specific instructions; the mnemonics following each landmark heading correspond to one executable statement. Note that the “+ n” offsets used in the landmark headings refer to executable statements; for example:

```
STATEMENT LABEL 200 + 2 <-- second executable statement
```

Example 9-3. Compiler Listing—ICODE Listing

```
            OBJECT CODE WITH INSTRUCTION MNEMONICS
                     START OF PROGRAM UNIT
000000 7 CON %000401 7 CON %000016 7 CON %000000
000003 7 CON %000016 7 CON %000000 7 CON %000026
000006 7 CON %000026 7 CON %030000 0 7 CON %037400 ?
000011 7 CON %007001 7 CON %032405 5 7 CON %104404
000014 7 CON %032405 5 7 CON %100413 7 CON %032405 5
000017 7 CON %105006 7 CON %000212 7 CON %003402
000022 7 CON %002000 7 CON %000401 7 CON %000056 .
000025 7 CON %177777 7 CON %000056 . 7 CON %177777
000030 7 CON %000036 7 CON %000036 7 CON %030114 0L
000033 7 CON %067564 ot 7 CON %020043 # 7 CON %047567 Ow
000036 7 CON %067145 ne 7 CON %071047 r' 7 CON %071440 s
000041 7 CON %047141 Na 7 CON %066545 me 7 CON %053141 Va
000044 7 CON %066165 lu 7 CON %062524 eT 7 CON %060570 ax
000047 7 CON %020103 C 7 CON %064141 ha 7 CON %071147 rg
000052 7 CON %062400 e 7 CON %037400 ? 7 CON %007001
000055 7 CON %032405 5 7 CON %037400 ? 7 CON %007405
000060 7 CON %032404 5 7 CON %037400 ? 7 CON %012014
000063 7 CON %032405 5 7 CON %037400 ? 7 CON %020005
000066 7 CON %032404 5 7 CON %037400 ? 7 CON %022412 %
000071 7 CON %002000 7 CON %000000 7 CON %000002
000074 7 CON %000004 7 CON %000006 7 PCAL 000
000077 7 ADDS +002 0 LWP -007 1 LWP -007
000102 2 LWP -007 3 LWP -007 7 PUSH 733
000105 0 LADR L+016 7 STAR 7 0 STRP 0
000110 0 LLS 01 7 STAR 6 0 LDRA 6
000113 7 PUSH 700 7 ADDS +014
       STATEMENT LABEL 15 LINE # 9.000
000115 0 LADR S-000 7 STOR L+002 7 ADDS +012
000120 0 RDP 0 ADDI -076 1 LDI +006
000123 7 PUSH 711 0 LDLI +100 1 LADR L+001
000126 7 PUSH 711 7 ADDS +002 0 LDI +074
000131 7 PUSH 700 0 PCAL 000 7 STRP 7
000134 7 ADDS +014 0 LDLI +200 1 LADR L+001
000137 7 PUSH 711 7 ADDS +002 0 LDI +014
000142 7 PUSH 700 0 PCAL 000 1 LADR L+002,I
000145 0 SETS 7 STRP 7
       STATEMENT LABEL 20 LINE # 10.000
000147 0 LDI +001 0 BUN +174,I
       STATEMENT LABEL 20 + 1 LINE # 11.000
000151 0 LADR S-000 7 STOR L+002 7 ADDS +013
000154 0 LDI +005 7 PUSH 700 0 LDLI +160
000157 0 ORRI +001 1 LADR L+001 7 PUSH 711
000162 7 ADDS +002 0 LDI +034 7 PUSH 700
000165 0 PCAL 000 7 STRP 7 0 LADR L+013
000170 1 LDI +002 2 LDI +002 3 LDI +001
000173 7 PUSH 733 7 ADDS +010 0 LDLI +360
000176 0 ORRI +001 1 LADR L+001 2 LADR S-013
```

You can also generate an ICODE listing, either interactively or from a command file, using Inspect or Binder. The following Binder command displays the object code in ICODE format from all code blocks in the object file MYOBJ:

```
DUMP CODE * ICODE FROM myobj
```

For more information about the DUMP command, see the *Binder Manual*.

You can view the code in a running process or in an Inspect save file using the Inspect ICODE command. The following Inspect command lists in ICODE format the machine instructions for the first four FORTRAN statements in a subprogram named MYSUB:

```
ICODE at #MYSUB FOR 4
```

For more information about the ICODE command, see the *Inspect Manual*.

CROSSREF Listing

If the CROSSREF option is in effect, SYMSERV collects cross reference data for FORTRAN's output listings. The first page of the map lists the source files in the compilation. Subsequent pages list the cross reference for specified identifier classes.

For more information, see the *CROSSREF Manual*.

LMAP Listing

If the LMAP option is in effect, BINSERV prints a map of all the procedures included in the object file. A second map of all common data blocks is included in the output. Table 9-3 shows the categories for the LMAP code block listing for a routine named NAME and Table 9-4 on page 9-18 shows the categories for the LMAP data block listing.

Table 9-3. LMAP Code Block Listing (page 1 of 2)

Name	Meaning
PEP	Code-relative octal address of the entry in the Procedure Entry Point (PEP) table for NAME
BASE	Code-relative octal address of the first word of NAME
LIMIT	Code-relative octal address of the last word of NAME
ENTRY	Code-relative octal address of the entry point of NAME; that is, the address where execution begins for that entry point
ATTRS	The attributes of NAME, where M is the name of the main program unit E is the name of a secondary entry point V indicates that NAME has a variable number of parameters.
M	is the name of the main program unit
E	is the name of a secondary entry point
V	indicates that NAME has a variable number of parameters
NAME	Name of a user-written or FORTRAN-supplied program unit, entry point, or library procedure that is bound into the object file by BINSERV
DATE	The date NAME was compiled

Table 9-3. LMAP Code Block Listing (page 2 of 2)

Name	Meaning
TIME	The time of day NAME was compiled
LANGUAGE	The source language of NAME
SOURCE FILE	The name of NAME's source file

Table 9-4. Data Block Listing

Name	Meaning
BASE	Lowest address of the data block. Either the G-relative address (six octal digits) of the first word of the block, or the extended address (ten octal digits) of the first bytes of the block.
LIMIT	Highest address of the data block. Either the G-relative address (six octal digits) of the last word of the block, or the extended address (ten octal digits) of the last byte of the block.
TYPE	The data block type: COMMON, OWN, or SPECIAL.
MODE	The allocation characteristic of the block. WORD indicates that the block must begin on a word (even-numbered byte) boundary, and STRING means that the block cannot be in the upper half of the user data segment.
NAME	The symbolic name of the data block.
DATE	The date of compilation of the object file from which the block was obtained.
TIME	The time of day of the compilation.
LANGUAGE	The source language of the procedure that declares the data block.
SOURCE FILE	The file name of that procedure's source file.

The map is in alphabetic order by name. If you specify the LOC option for the LMAP listing, BINSERV displays an additional map that shows the same data listed in ascending order of base addresses.

The LMAP * option also requests entry point and common block cross-reference lists in addition to the maps ordered by name and base address. Note that maps for entry points and for data blocks are listed.

To find the actual address of an item in your program, you must add any code-relative address (format specification or statement label in a program unit map, or a code offset in a CODE or ICODE listing) to the BASE address for that program unit. You can use this information to test, debug, and monitor the object program.

Completion Message

BINSERV and FORTRAN both print statistics after an object file is built. If the SYNTAX directive is in effect, only FORTRAN statistics are listed. Numeric values are displayed in base ten.

The BINSERV process displays the following statistics:

- Name of object file created as a result of compilation
- Total number of errors reported by BINSERV
- Total number of warnings reported by BINSERV
- Total number of words of code area needed; this includes separate listings for:
  - PEP size
  - Read-only arrays
  - Storage occupied by program units
  - Gap size at 32K boundary
  - XEP size
- Total number of words of primary global storage needed
- Total number of words of secondary global storage needed
- Number of (virtual) memory code pages to be allocated for the program at run time
- Number of (virtual) memory data pages to be allocated for the program at run time

If you use the SYNTAX directive or the compiler detects a fatal error in the program, FORTRAN displays the message

```
No object file created
```

before listing statistics from the compilation of source code.

Following these messages, FORTRAN lists these statistics from the compilation of the source code:

- Total number of error messages issued by the compiler
- Total number of warning messages issued by the compiler
- Total number of words of primary global storage used
- Total number of words of secondary global storage used
- Number of words needed by the compiler for its symbol table
- Number of source lines in the compilation
- Elapsed time for the compilation including cross-reference operation if CROSSREF is in effect

BINSERV does not include the XEP size in the code-area size. Also, note that FORTRAN's global storage size might differ from BINSERV's: FORTRAN counts only declared data; BINSERV includes run-time data structures in the global storage size.

Example 9-4. Compiler Listing—Completion Message

```
BINDER - OBJECT FILE BINDER - T9621D10 - (08JUN92)      SYSTEM \USERS
Copyright Tandem Computers Incorporated 1982-1992
Object file \USERS.$TOOLS.FTNTOOLS.MYOBJ
TIMESTAMP 1992-09-05 23:06:05
      47 Code pages
       3 Primary data words
     795 Secondary data words
       4 Data pages
       0 Resident code pages
       0 Extended data pages
     798 Top of stack location in words
       1 Code segment
       0 Binder Warnings
       0 Binder Errors
       0 Compiler errors
       0 Compiler warnings
       3 Primary data words
       8 Secondary data words
    2880 Maximum symbol table size in words
      43 Lines of source text
 0:00:59 Elapsed time
```

Compiler Termination Codes

When the compilation ends, FORTRAN sends one of the following completion codes to TACL:

Completion Code	Meaning
0	No errors or warnings.
1	No errors; one or more warnings. An object file was created.
2	Fatal errors; no object file was created.
3	Compilation did not complete. There may be an incomplete object file or none at all. Possible reasons for this are: ● Your processor is not licensed for FORTRAN ● A table overflowed ● A serious I/O error occurred
5	Internal compiler error; contact your HP analyst.
8	One or more warning messages; an object file was created but its name was changed. See the last page of the compiler listing for the name.

You can use TACL to check completion codes, but this is generally useful only if you are doing batch processing. In most cases, the simplest procedure is to check the compiler listing to find out what happened.

Separate Compilation

You can use Binder to bind together object files created from separate runs of the FORTRAN compiler in addition to object files created by C, COBOL85, PASCAL, and TAL compilations. Binder also allows you to examine, combine, or modify object files. It operates as either of two processes:

- BINSERV, the compile-time binder, which is driven by compiler directives
- BIND, which is driven by commands that you enter interactively or that it reads from a file

Compilation Unit

A compilation unit consists of all the input to a single run of the compiler. A compilation unit can include any number of program units in the compiler's input source file and additional source files named in SOURCE directives.

FORTRAN organizes the compiled object code into blocks of code and data that BINSERV uses to build the object file.

BINSERV can also include copies of previously compiled object code in the new object file that it creates. This happens if the compilation unit includes a SEARCH directive that names object files from which BINSERV can retrieve code or data to satisfy references to other program units or to common data. These are called external references.

The output object file from a binder process is called a target file. If the target file includes a main program unit, the file is an executable program file. An object file can have only one main program unit.

If you plan to use the compile time binder, see Section 10, Compiler Directives, for a detailed description of the directives that affect building of the object file: SEARCH, LMAP, COMPACT, and SYNTAX.

If you want to use Binder as a separate process, see the *Binder Manual*.

Code Blocks and Data Blocks

Blocks are the smallest relocatable (bindable) units of code or data. For FORTRAN, these rules apply:

- Each program unit results in a separate code block. FORTRAN uses the name given in the PROGRAM, SUBROUTINE, or FUNCTION statement to identify the program unit's code block. If you omit the PROGRAM statement, FORTRAN uses the default name MAIN^ for the main program unit. A block data subprogram does not have a code block.
- A program unit can have up to two OWN data blocks containing local data items that are statically allocated as a result of DATA and SAVE statements. Each block name has the same name as its program unit. The name is preceded by a plus

sign (+) if it is in the user data segment or by an ampersand (&) if it is in the extended data segment.

- An executable program can have any number of COMMON data blocks shared by the program units that declare them in one or more source program COMMON statements. Each block name is the name that appears between slashes in COMMON statements (FORTRAN uses BLANK^ for an unnamed common block), preceded by a period (.) if it is in the user data segment or by a dollar sign ($) if it is in the extended data segment.
- The compiler also creates SPECIAL data blocks. These require special handling by the Binder, such as allocating them to specific addresses or having their contents merged from same-named blocks in multiple object files. Every SPECIAL data block has a pound sign (#) in its name. FORTRAN creates some data blocks only in the OLD environment, others only in the COMMON environment, and still others in either environment. The SPECIAL data blocks are shown in Table 9-5.

Table 9-5. FORTRAN Data Blocks (page 1 of 2)

Block Name	Environment	Contents
COMMON#POINTERS	Both	The indirect addressing pointer word for data items in user data segment common blocks.
$EXTENDED#STACK	Both	The indirect addressing pointer word for data items in extended segment common blocks.
EXTENDED#STACK#FRAME and EXTENDED#STACK#POINTERS	Both	Used for addressing the extended stack area.
#FLUT	OLD	The FORTRAN Logical Unit Table. It contains the address of each File Control Block in the PUCB, indexed by FORTRAN I/O unit number.
#G0	OLD	The first three words of the user data segment. It contains information about the #RUCB block.
#HIGHBUF	OLD	The area reserved for file buffers in the upper half of the user data segment.
#LOWBUF	OLD	The area reserved for saved messages, $RECEIVE queues, and file buffers in the lower half of the user data segment.
#PUCB	OLD	The Program Unit Control Block. It contains language-specific information, and includes a File Control Block for each FORTRAN I/O unit defined for the program.

Table 9-5. FORTRAN Data Blocks (page 2 of 2)

Block Name	Environment	Contents
#RUCB	OLD	The Run-Unit Control Block. It contains pointers to most other special data blocks.
#MCB	COMMON	Global data area for run-time environment.
#CRE_HEAP	COMMON	Global data area for run-time environment.
#CRE_GLOBALS	COMMON	Pointers to data.
#RECEIVE	COMMON	The buffer that holds messages received from $RECEIVE.
##FT nnn	COMMON	File Control Blocks.
Common Blocks	Both	User data declared in COMMON statements.

Many interactive Binder commands require parameters specified as block names. All the names of blocks in an object file are listed in the load maps that result from compilation or interactive binding.

An object file can contain blocks derived from multiple languages. You can use COBOL85, FORTRAN, and TAL compiled code in a single program. For additional information about mixed-language programming, see Section 13, Mixed-Language Programming.

Compiling Programs That Use Extended Data Space

User data space contains up to 64K words. Section 12, Memory Organization, explains how FORTRAN allocates storage within the user data space. If your program requires additional data space, you can use compiler directives to tell FORTRAN to store some of your program data in extended data space.

To use extended data space in a program that you compile in a single compilation:

- Use LARGECOMMON directives for the common data blocks you want FORTRAN to store in extended memory.
- Use LARGEDATA directives for the non-common variables, arrays, and RECORDs you want FORTRAN to store in extended memory.

To use extended data space in a program that you compile in several separate compilations:

- Follow the same steps listed above, but make sure that any common block listed in a LARGECOMMON directive in one compilation is also listed in a

LARGECOMMON directive in every compilation that includes the same common block.

- Include an EXTENDEDREF compiler directive at the beginning of every compilation that does not include a LARGECOMMON directive or a LARGEDATA directive.
- Use the LARGESTACK directive (or the BIND command SET LARGESTACK) if the extended stack area is too small.

FORTRAN can handle programs that use up to 128 megabytes of extended data space, but to execute such programs, you must have enough free disk space on your system to allocate a contiguous block of virtual memory for the data space.

It takes more machine instructions to manipulate 32-bit addresses than 16-bit addresses. Programs with a large amount of data in extended memory are larger and slower than they would be otherwise. Therefore, smaller and more frequently used data items should be kept in the user data segment, while larger and less frequently used data should be moved to extended memory.

Binding Programs That Use Extended Memory

Binder cannot produce object files that combine FORTRAN program units compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA directives and FORTRAN program units compiled without EXTENDEDREF, LARGECOMMON, or LARGEDATA directives.

If you only use the compile-time binder, you don’t have to worry about this distinction unless you include SEARCH directives in your program. Since all program units from a single compilation use the same extended-memory options, FORTRAN program units from a single compilation are always bindable.

If you use SEARCH directives or stand-alone Binder to combine program units from different compilations, you must make sure that all program units you attempt to bind into a single object file are compiled with extended-memory options (that is, with LARGECOMMON, EXTENDEDREF, or LARGEDATA directives)—or that all program units you attempt to bind are compiled without extended-memory options.

User Library Alternatives for Utility Subprograms

The user library feature of the operating system is a convenient way to make packages of general-utility subprograms available to multiple application programs. Alternative methods are:

- Use the SEARCH directive when you compile the application programs, causing each application program's object file to include its own copy of the utility subprograms that it needs. The disadvantages with SEARCH are that the application program object files are larger than they should be, and bug fixes and other improvements to the utility subprograms are not propagated into all of application programs unless they are re-compiled or at least re-bound.
- Use SYSGEN to incorporate the subprogram package into the system library code space. The disadvantage with SYSGEN is that you must request your system manager to do this.

For more information on libraries, see the LIBRARY compiler directive in Section 10, Compiler Directives, and the FORTRAN RUN command in Section 11, Running and Debugging Programs.

Sample Programs Using the Search Directive

This section contains two sample programs. One shows development of a program using SEARCH and SOURCE directives for subprograms. It also includes an alternative development cycle using the interactive Binder.

The second program uses SEARCH to select certain subroutines from an object file of subroutines, some of which also contain subprogram calls. (The calls can be recursive or call other subroutines.) It shows how different users can use a subroutine object file as a common resource.

Using the SEARCH Directive—Sample Program 1

This sample program transposes the rows and columns of a five-row, five-column matrix named ARRAY. The steps in the development of the program are flexible. That is, coding and compilation can take place in different sequences. However, use of the SEARCH directive is effective only if compiled code is available to satisfy external references; this implies that you cannot use SEARCH from a subprogram for a main program unit.

1. Code and compile separately a subroutine subprogram that is named PRINTARRAY. The resulting code and data blocks are in an object file named PRINTO. (The source file name is PRINTS.)
2. Code a dummy for the transpose subroutine, TRANSPOSE. The source file name is DUMMYS. The dummy subroutine allows program testing to occur in stages.

3. Code and compile the main program unit. The source file includes a SOURCE directive for the DUMMYS file. A SEARCH directive identifying PRINTO as the search list is included to have BINSERV include the object code from Step 1 in the program file. The program file is named MAKEONE. It contains copies of the MAIN^, PRINTARRAY, and TRANSPOSE dummy code.
4. Test run the program produced in Step 3. (This step is completely independent of Step 5.)
5. Code the actual transpose subroutine, TRANSPOSE. The source file is named XPOSE.
6. Based on the output of Step 5, alter the source code for the main routine. Include a SOURCE directive for XPOSE (for the actual transpose subroutine). Include a SEARCH directive identifying MAKEONE (to get the code and data blocks for PRINTARRAY). Recompile the main routine, deleting the old MAIN file. (The new version MAIN^ object code will then be in a file named MAIN.) The program file is named MAKETWO.

Each program unit that declares ARRAY uses the source file GLOBAL for the declarations. (The SOURCE directive for GLOBAL is part of each source file except the dummy for the TRANSPOSE code.) GLOBAL contains these declarations for ARRAY:

```
INTEGER array, asize
PARAMETER (asize = 5)
COMMON /array/ array (asize, asize)
```

Following are source listings for the program.

Step 1—PRINTARRAY Subprogram

Compilation of this source file results in an object file that contains a code block named PRINTARRAY (the name on the SUBROUTINE statement). The object file has the following data blocks: #FLUT, #G0, #LOWBUF, #PUCB, #RUCB, ARRAY and COMMON#POINTERS. #G0 contains a pointer to #RUCB. (FORTRAN allocates no storage for the local variables in the object file. It allocates this storage at execution time.)

```
      SUBROUTINE printarray
?SOURCE global
      INTEGER row, column
      WRITE(UNIT=6,FMT='(1h )')
      DO 10 row = 1, asize
      WRITE(6,1000) (array(row,column), column=1, asize)
10 CONTINUE
      RETURN
1000 FORMAT(10I10)
      END
```

Step 2—TRANSPOSE Dummy Subprogram

The following subprogram is compiled using the SOURCE directive from the main:

```
SUBROUTINE transpose
RETURN
END
```

If separately compiled, the dummy's object file would consist of a code block named TRANSPOSE. No common blocks would be included. (The special blocks #FLUT, #G0, #LOWBUF, #PUCB, and #RUCB are always included in object files.)

Step 3—Main Program, Version One

Compilation of the source file shown below results in an object file containing a code block named MAIN^ (the compiler's default name). It also contains PRINTARRAY, and TRANSPOSE.

```
?SEARCH printo
?SOURCE dummys
?SOURCE global
      INTEGER row, column
C     Initialize the array --
      DO 20 row = 1, asize
         DO 10 column = 1, asize
           array(row, column) = asize*(row-1) + column-1
      10 CONTINUE
      20 CONTINUE
C      Display the initialized array
         CALL printarray
C      Transpose the array
         CALL transpose
C      Display the transposed array
         CALL printarray
         STOP 'End of example'
         END
```

Step 4—TRANSPOSE Subprogram

The object file resulting from compilation of this code has a code block, TRANSPOSE. The file also contains the data blocks seen in earlier steps, including ARRAY. The local data variables are allocated at execution time.

```
C     This is the real transpose subroutine.
      SUBROUTINE transpose
?SOURCE global
      INTEGER temp, i, j

      DO 20 i = 2, asize
        DO 10 j = 1, i-1
         temp = array(j, i)
         array(j, i) = array(i, j)
         array(i, j) = temp
   10   CONTINUE
   20 CONTINUE
      RETURN
      END
```

Step 5—Version One —Program Test

Run the program. The output produced begins with zero (rather than one) as the first subscript value. You must change the calculation in the main program to get the planned results.

The work on TRANSPOSE is not dependent on the calculations in the main program; therefore, you can code it concurrently with the development and testing of other program units.

Step 6—Main Program, Version Two

The source file for MAIN is altered for the required calculation. Instruct BINSERV to purge the old object file and use the same file name for the target file. (The FORTRAN implied run command has the same effect as when the compiler built object files.)

The source file includes a SOURCE directive for the actual TRANSPOSE subroutine. FORTRAN compiles the code and passes it to BINSERV. Since BINSERV has a code block named TRANSPOSE for the target file (from the compiler), there is no external reference to TRANSPOSE remaining when BINSERV executes the SEARCH directive.

Therefore, the dummy TRANSPOSE is ignored.

Since no data was changed, there is no need to recompile PRINTARRAY.

```
?SEARCH makeone
?SOURCE xpose
?SOURCE global
C     Initialize the array --
      DO 20 row = 1, asize
        DO 10 column = 1, asize
         array(row, column) = asize*(row-1) + column
   10   CONTINUE
   20 CONTINUE

C     Display the initialized array
      CALL printarray

C     Transpose the array
      CALL transpose

C     Display the transposed array
      CALL printarray
      STOP 'End of example'
      END
```

Alternative Development Using the Interactive Binder

For the descriptions of the ADD and BUILD commands used to perform the following steps, see the *Binder Manual*.

1. Code and compile separately three routines: the main program (given the name MAIN^ by FORTRAN), the subroutine PRINTARRAY, and a dummy for the transpose subroutine, TRANSPOSE. The object files are in disk files named MAIN, PRINT, and DUMMY.
2. Use the interactive Binder to build a program file by binding the object code produced in Step 1. The file name for the program file is MAKEONE. It contains copies of the MAIN^, PRINTARRAY, and TRANSPOSE code. No changes are made to the MAIN, PRINT, and DUMMY disk files.
3. Run the program produced in Step 2.

4. Based on the output of Step 3, alter the source code for the main routine. Recompile the main routine, and delete the old MAIN disk file. (The new MAIN^ object code will also be in a file named MAIN.)
5. Use the interactive Binder to build a new program file. Input to this step is the program file MAKEONE and the new MAIN file. Since the command to build the new program file specifies a different file name (MAKETWO), Binder does not purge MAKEONE. MAKETWO contains copies of PRINTARRAY and TRANSPOSE from MAKEONE along with the new MAIN^ routine from MAIN.
6. Code and compile the actual transpose subroutine, TRANSPOSE. The disk file is named XPOSE.
7. Use the interactive Binder to build the final program file, LASTMAKE, by adding the actual TRANSPOSE code and deleting the dummy subroutine.

Using the SEARCH Directive—Sample Program 2

This program is a pen-primer routine for use with a commercially available plotter. The plotter source code includes a file of subroutines (not shown here). The pen primer uses a subset of the subroutines. Using the SEARCH directive, compilation results in the inclusion of only those code and data blocks that are needed to satisfy external references. You do not need to use a SOURCE directive to recompile subroutine source code.

```
C Main Program -- Pen Primer
?UNIT (8)
?SEARCH fpslib
      COMMON/termio/kebord, icrt
      COMMON/zdonz/ldevzi, ldevzo
      CHARACTER*1 mess(16)
      DATA mess/'P','R','I','M','I','N','G',' ',
     + 'T','H','I','S',' ','P','E','N'/
      kebord = 4
      icrt = 4
      ldevzi = 8
      ldevzo = 8
      OPEN(8,FILE='$S.#PLOTS',STATUS='OLD',SPACECONTROL='NO',
     + IOSTAT=ioerr,ERR=8888)
      CALL plots(53,0,8 )
      CALL newpen (1)
      CALL symbol(2.,5.,.2,mess,0.,16)
      CALL symbol(2.,4.5,.2,mess,0.,16)
      CALL newpen (2)
      CALL symbol(2.,4.,.2,mess,0.,16)
      CALL symbol(2.,3.5,.2,mess,0.,16)
      CALL newpen (3)
      CALL symbol(2.,3.,.2,mess,0.,16)
      CALL symbol(2.,2.5,.2,mess,0.,16)
      CALL newpen (4)
      CALL symbol(2.,2.,.2,mess,0.,16)
      CALL symbol(2.,1.5,.2,mess,0.,16)
      CALL plot(8.5,0.,999)
      WRITE (icrt,1)
      1 FORMAT(' ALL 4 PENS PRIMED'/)
      STOP
 8888 WRITE (4,8889) ioerr
```

```
8889 FORMAT(' FILE ERROR $PLOTS: ',I3)
     STOP
     END
```

10 Compiler Directives

You use compiler directives to specify and control many aspects of a compilation, such as:

- To specify listing features
- To specify alternate source files
- To enable run-time bounds checking
- To control data allocation
- To specify data area size
- To control how the $RECEIVE file is used at run-time
- To control the building of object files
- To specify the line length for source input files
- To declare procedures not written in FORTRAN
- To specify user libraries
- To save compiler directive values

Table 10-1. Summary of Compiler Directives (page 1 of 4)

Directive	Action
ABORT	Specifies compiler action if a file named in a SOURCE or CONSULT directive cannot be opened. Default is ABORT.
ANSI	Treats columns 73 through 132 as comments. Default is NOANSI.
BOUNDSCHECK	Compiler generates code to verify array subscripts at run-time. Default is NOBOUNDSCHECK.
CODE	Lists octal instruction codes for each program unit, following source listing. Default is NOCODE.
COLUMNS	Defines the line length of records in a source file. Default is COLUMNS 132.
COMPACT	Attempts to compact code space of object code. Default is NOCOMPACT.
CONSULT	Declares procedures not written in FORTRAN.
CROSSREF	Generates cross-reference information for selected identifier classes.
DATAPAGES	Specifies the number of virtual memory pages to allocate for data storage in the user data segment.
ENDIF	Terminates the effect of a preceding IF or IFNOT directive.
ENV	Specifies whether the process containing this program uses the C-series runtime library or the D-series run-time library. Default is ENV OLD.

Table 10-1. Summary of Compiler Directives (page 2 of 4)

Directive	Action
ERRORFILE	Stores compilation error messages in a file.
ERRORS	Sets the maximum number of errors for a compilation. Default is ERRORS 100.
EXTENDCOMMON	Uses indexed indirect addressing to access simple variables in common blocks. Default is NOEXTENDCOMMON.
EXTENDEDREF	Generates code that uses doubleword addresses for parameters in CALL statements and function references. Default is NOEXTENDEDREF.
FIXUP	Causes Binder processing of the object file to make it executable. Default is FIXUP.
FMAP	Includes a file map in the listing. Default is NOFMAP.
GUARDIAN	Declares procedures as Guardian procedures or as utility routines.
HIGHBUFFER	Allocates space for the run-time buffer pool in the upper half of the user data segment. Default is HIGHBUFFER 0.
HIGHCOMMON	Allocates common storage in upper data memory for specified common blocks.
HIGHCONTROL	IF ENV OLD is in effect, compiler allocates I/O control blocks in upper data memory. If ENV COMMON is in effect, compiler allocates #MCB in upper data memory. Default is NOHIGHCONTROL.
HIGHPIN	Specifies whether the process containing the program with this directive can be run at a PIN that is greater than 255. Default is NOHIGHPIN.
HIGHREQ	Specifies whether the process containing the program with this directive can be opened by a process running at a PIN that is greater than 255. Default is NOHIGHREQ.
ICODE	Lists the symbolic instruction codes for each program unit after the source listing. Default is NOICODE.
IF	Processes subsequent source lines if the specified toggle is set.
IFNOT	Processes subsequent source lines if the specified toggle is reset.
INSPECT	Selects Inspect as the default debugger. Default is NOINSPECT.
INTEGER	Specifies the size of all subsequent integer entities (whose size is not specified) in the source file. Default is INTEGER*2.
LARGECOMMON	Allocates specified common blocks in the extended data segment.
LARGEDATA	Allocates memory space in the extended data segment for local data.
LARGESTACK	Specifies the block size for dynamically allocated variables of LARGEDATA directives.
LIBRARY	Declares a file as a user library.

Table 10-1. Summary of Compiler Directives (page 3 of 4)

Directive	Action
LINES	Specifies the number of lines to write to the listing file before a page skip.
LIST	Controls listing of source lines; enables CODE, CROSSREF, ICODE, FMAP, LMAP, MAP, and PAGE. Default is LIST.
LMAP	Causes load maps to be listed after identifier map and cross-reference tables. Default is LMAP ALPHA.
LOGICAL	Specifies the size of all subsequent logical entities in source file. Default is LOGICAL*2.
LOWBUFFER	Allocates space for the run-time buffer pool in the lower half of the user data segment. Default is LOWBUFFER 512. LOWBUFFER has no effect if ENV COMMON is specified in the same compilation.
MAP	Lists table of data blocks and local identifiers for a program unit. Default is MAP.
NONSTOP	When present in the ‘main’ routine of a program that specifies ENV COMMON, specifies that the program is capable of running as a NonStop process. Default is NONONSTOP.
PAGE	Prints a specified page title on the next page of the listing file, and usually causes a page skip.
POP	Restores the previous values of one or more compiler directives.
PRINTSYM	Includes unreferenced identifiers in MAP listing. Default is NOPRINTSYM.
PUSH	Saves the current values of one or more compiler directives.
RECEIVE	Specifies values for parameters that control aspects of interprocess communication.
RESETTOG	Resets one or more specified toggles.
RUNNAMED	Specifies that the program containing this directive run as a named process. Default is NORUNNAMED.
SAVE	Saves specified information related to the starting environment of a process.
SAVEABEND	Creates a save file in case of abnormal program termination. Default is NOSAVEABEND.
SEARCH	Specifies list of object files to search for unsatisfied external references at compile time.
SECTION	Assigns a name to a section of a source file for use in a SOURCE directive.
SETTOG	Sets specified toggles for conditional compilation control.
SOURCE	Causes the compiler to read part or all of an alternate source input file.
SUBTYPE	Specifies the process subtype for an object file. Default is SUBTYPE 0.

Table 10-1. Summary of Compiler Directives (page 4 of 4)

Directive	Action
SUPPRESS	Lists only error messages and compilation statistics; overrides LIST. Default is NOSUPPRESS.
SYMBOLS	Includes a symbol table in the object file for use by Inspect. Default is NOSYMBOLS.
SYNTAX	Searches source file for syntax errors; does not produce object file.
UNIT	Causes a unit to exist; declares the properties of the file or files connected to the unit.
WARN	Lists warning messages regardless of the LIST setting. Default is WARN.

Using Compiler Directives

The general form for compiler directives is:

```
? directive [, directive ]...
```

?

in the first column of a source line designates that it is a compiler directive line.

directive

is one of the compiler directives described in this section.

Considerations

- FORTRAN ignores blanks in compiler directives, except within quoted strings.
- You can specify more than one directive per line. Separate directives with commas.
- You can continue directives that have a parameter list over multiple lines. The break in the parameter list can occur between parameters anywhere after the opening parenthesis. The continuation line must also begin with a question mark in column one. For example:

```
?ERRORS 100, FMAP, SOURCE (file1, file2,
?file3)
```

- You can use an equal sign (=) between any directive or option keyword and its numeric value, but it is not required. For example, ?DATAPAGES = 64 and ?DATAPAGES 64 are equivalent.
- You can use decimal or octal notation for all directives and options that require numeric values.

Observe the following restrictions in the placement of compiler directives:

- Specify the following directives as the first or only directive on a line:

  COLUMNS

  PAGE

- Specify the following directives as the only directive on a line:

  ENDIF

  SECTION

- Specify the following directives last on a directive line:

  IF

  IFNOT

  RESETTOG

  SETTOG

  SOURCE

- Specify the following directives either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement:

ABORT	FIXUP	INTEGER	NONSTOP
ENV	FMAP	LARGECOMMON	RUNNAMED
ERRORFILE	HIGHCOMMON	LARGESTACK	SUBTYPE
ERRORS	HIGHPIN	LIBRARY	SYNTAX
EXTENDCOMMON	HIGHREQ	LOGICAL	UNIT
EXTENDEDREF			

- Specify the following directives either with the FORTRAN command, or in the source input file preceding the first FORTRAN statement, or between the END line of one program unit and the first FORTRAN statement of the next program unit:

ANSI	CODE	MAP
BOUNDSCHECK	CROSSREF	SYMBOLS

- Specify the following directives either with the FORTRAN command (after the semicolon following the object file name) or anywhere in the source input file:

```
COLUMNS       ICODE       LOWBUFFER     SAVEABEND
COMPACT       IF          PAGE          SEARCH
CONSULT       IFNOT       POP           SECTION
DATAPAGES     INSPECT     PRINTSYM      SETTOG
ENDIF         LARGEDATA   PUSH          SOURCE
GUARDIAN      LINES       RECEIVE       SUPPRESS
HIGHBUFFER    LIST        RESETTOG      WARN
HIGHCONTROL   LMAP        SAVE
```

ABORT Compiler Directive

The ABORT directive specifies whether the compiler should abort if it cannot open a file referenced in a SOURCE or CONSULT compiler directive.

Use the ABORT directive if you run the compiler without a home terminal or if your home terminal is unattended—for example, if you use the NetBatch product or a TACL macro for multiple compilations, or you compile a large program from an unattended terminal.

The default value is ABORT.

```
[NO]ABORT
```

Considerations

- Specify the ABORT directive either on the FORTRAN command line following the semicolon after the object file name or in the source input file before the first FORTRAN statement.
- If there are two or more properly placed ABORT directives, the compiler obeys the last one processed at the time it fails to open a SOURCE or CONSULT file.
- If an ABORT directive appears after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- The ABORT directive applies only if the compiler cannot open a source file specified in a SOURCE directive or an object file specified in a CONSULT directive.

- If you specify NOABORT and the compiler cannot open a SOURCE or CONSULT file, the compiler displays the following on the home terminal:

```
File file-name not in directory.


Enter ?SAME to try same file.
Enter ?ABORT to abort compile.
Enter ?IGNORE to skip file.
Otherwise enter another file name.


Enter response:
```

  If a file system error code (including EOF) is returned or you enter ?ABORT, the compiler calls ABEND with completion code 3 (premature termination). If you enter ?SAME, or ?IGNORE, or a file name, the compiler proceeds as directed.

- If FORTRAN is unable to open a SOURCE or CONSULT file, and an ABORT directive is in effect, the compiler does not display a message on the home terminal. Instead, it writes the following message to its OUT file and ABENDs with completion code 3:

```
OPEN OF directive FILE FAILED
FILE MANAGEMENT ERROR # n: file-name
```

Example

```
?ABORT
```

ANSI Compiler Directive

The ANSI directive instructs the compiler to ignore characters beyond position 72 of a source line and to pad each line that is shorter than 72 characters with blanks.

The default value is NOANSI.

```
[NO]ANSI
```

Considerations

- Specify ANSI if you need to conform to the ANSI standard line length. Note that the ANSI directive affects only the length of source records read by the compiler. It has no effect on other HP extensions to ANSI standard FORTRAN.

  Line length can also be specified with the COLUMNS directive. For more information, see the COLUMNS Compiler Directive on page 10-9.

- If you omit this directive, the compiler reads all characters in a source line.

- Specify the ANSI directive either with the FORTRAN command, or in the source input file preceding the first FORTRAN statement, or between the END line of one program unit and the first FORTRAN statement of the next program unit.
- The ANSI compiler directive is equivalent to COLUMNS 72, with the following exceptions:
  - If you specify ANSI, the compiler truncates or pads (with blanks) each source line to make it exactly 72 characters. The COLUMNS directive makes the compiler ignore all source text beyond the specified number of characters, but it does not affect source lines shorter than 72 characters. This distinction can be important if any character constants are continued from one line to the next.
  - The SOURCE and SECTION directives do not affect the ANSI mode.
- The ANSI directive temporarily overrides the effects of any COLUMNS directives.
- When an ANSI directive is in effect, the compiler prints each source-line image with a vertical bar character between the last character of a line and the first ignored character of that line. In other words, the compiler prints a vertical bar between columns 72 and 73 of each source-line image.

Example

```
?ANSI
```

BOUNDSCHECK Compiler Directive

The BOUNDSCHECK directive causes FORTRAN to generate code that verifies at run time that all array subscripts are within the lower and upper bounds declared for arrays dimensioned in the program.

The default value is NOBOUNDSCHECK.

```
[NO]BOUNDSCHECK
```

Considerations

- Array bounds violations cause arithmetic overflow at execution time. If you specify NOBOUNDSCHECK or omit this specification, the compiler does not perform any verification.
- You must compile each program unit in its entirety either with bounds checking or without.
- BOUNDSCHECK decreases program performance because of the additional code required to check each reference to an element of an array.
- Specify the BOUNDSCHECK directive either with the FORTRAN command, or in the source input file preceding the first FORTRAN statement, or between the END

line of one program unit and the first FORTRAN statement of the next program unit.

Example

```
?BOUNDSCHECK
```

CODE Compiler Directive

The CODE directive instructs the compiler to list the octal instruction codes generated for each program unit, following the source listing for that program unit.

The default value is NOCODE.

```
[NO]CODE
```

Considerations

The effect of the CODE directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.

Specify the CODE directive either with the FORTRAN command, or in the source input file preceding the first FORTRAN statement, or between the END line of one program unit and the first FORTRAN statement of the next program unit.

Example

```
?LIST, CODE
```

COLUMNS Compiler Directive

The COLUMNS directive causes the compiler to treat all text as comments beyond a specified column in each source line, beginning with the line that contains the COLUMNS directive.

This directive is designed to make it easy for a SOURCE file to bring in other files with different column conventions.

The default is COLUMNS 132.

```
COLUMNS number
```

number

is an unsigned integer in the range 12 through 132. The default value is 132. If you specify a value smaller than 12, FORTRAN issues a warning and uses 12. If you specify a value larger than 132, FORTRAN issues a warning and uses 132.

Considerations

- The COLUMNS directive can appear on the FORTRAN command line following the semicolon after the object file name. It can also appear anywhere in a source input file. A compilation can have any number of COLUMNS directives.

  When source files contain a COLUMNS directive, it must be the first or only directive on a line, and it must appear before the first SECTION directive if there are SECTION directives in the file.

- Use of the COLUMNS directive

  The COLUMNS value in effect at any given time depends on the context, as follows:

  - At the beginning of the main input file, the COLUMNS value is set by the last COLUMNS directive on the FORTRAN command line. If the FORTRAN command line has no COLUMNS directive, the COLUMNS value is 132.
  - A file read in by a SOURCE directive initially assumes the current COLUMNS value of the file that contains the SOURCE directive.
  - At the beginning of each section (of the main input file or of a file read in by a SOURCE directive), the current COLUMNS value is set by the last COLUMNS directive before a SECTION directive.

    (The COLUMNS directive is the only exception to the rule that FORTRAN ignores all directives or statements appearing outside a section specified in a SOURCE directive.)
  - Within each section, the current COLUMNS value is changed by any COLUMNS directive included in the section. Each COLUMNS directive is in effect only until the next COLUMNS or SECTION directive, the end of the SOURCE directive, or the end of the file, whichever comes first.
  - When a SOURCE directive is completed—that is, when all sections named in the SOURCE directive have been read or when the end of the file is reached, whichever comes first—the current COLUMNS value is restored to what it was when the SOURCE directive was encountered.
  - For cases not specified in this list, the current COLUMNS value is the value set by the most recently processed COLUMNS directive.

- If a source file has comments (such as source line identification) at the ends of the lines, place a COLUMNS directive at the beginning of the file.
- If a source file has no comments at the ends of the lines, specify COLUMNS 132 at the beginning of the file to prevent lines from being truncated if this file is read in by a file that has a smaller COLUMNS value.
- The ANSI compiler directive is equivalent to COLUMNS 72, but with the following differences:

- In ANSI mode, the compiler truncates or pads (with blanks) each source line to make it exactly 72 characters. The COLUMNS directive makes the compiler ignore all source text beyond the specified number of characters, but it does not affect source lines shorter than that. This distinction can be important if any character constants are continued from one line to the next.
- The SOURCE and SECTION directives do not affect the ANSI mode.

- The ANSI directive temporarily overrides the effects of any COLUMNS directives.
- When the value set by COLUMNS is less than 132, the compiler prints each source line image with a vertical bar character between the last character of a line and the first ignored character of that line. For example, when COLUMNS 72 is in effect, the compiler prints a vertical bar between columns 72 and 73 of each source line image.

Examples

This directive appears at the beginning of a source file that has comments beginning at column 81 of each line:

```
?COLUMNS 80
```

This directive appears at the beginning of a source file that has no comments at the ends of the lines. It prevents truncation of lines if this file is read in by a file that has narrow lines specified.

```
?COLUMNS 132
```

COMPACT Compiler Directive

The COMPACT directive specifies that BINSERV should attempt to compact the code space of the target file.

The default value is NOCOMPACT.

```
[NO]COMPACT
```

Considerations

A run unit can include as many as 32 code segments: 16 in user code space and 16 in user library space. Each code segment consists of up to 64K words. The FORTRAN compiler generates code-space blocks that cannot straddle the 32K-word boundary in a 64K-word code segment.

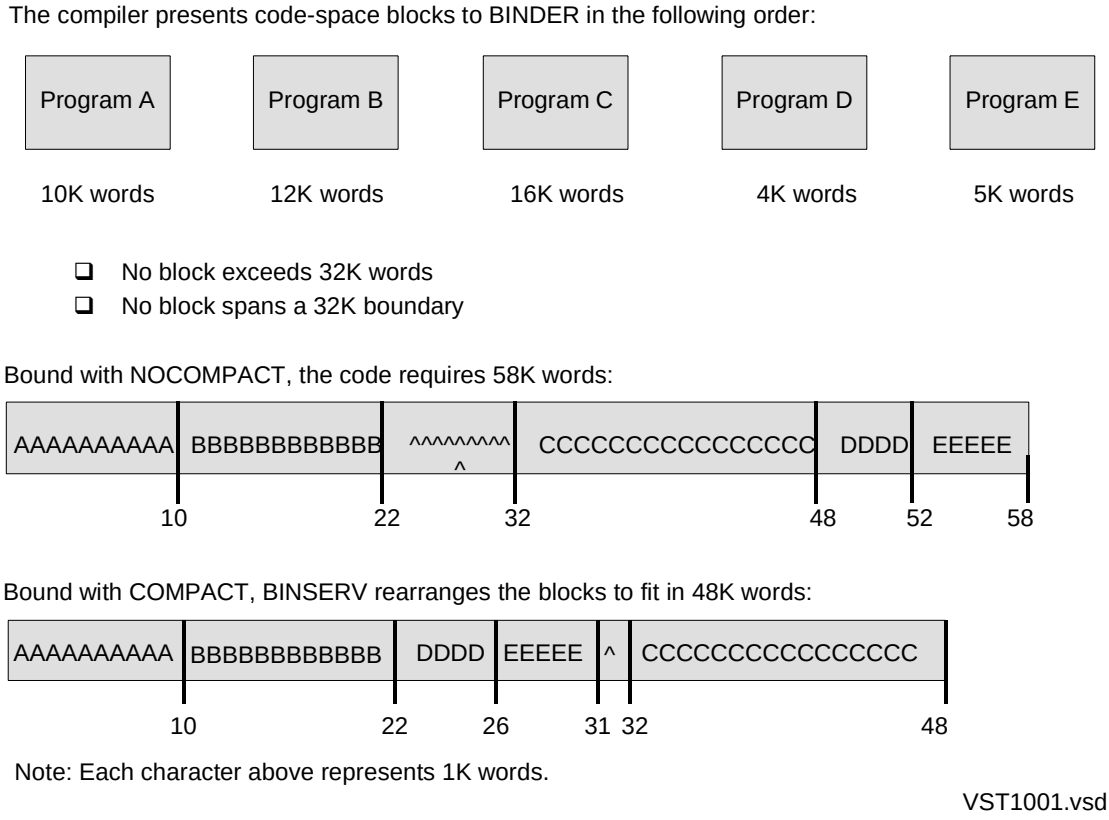
- If you specify NOCOMPACT, BINSERV allocates code-space blocks in the order in which they are presented to the compiler. This can leave gaps between the 32Kword boundary and the last code-space block below it, and between the 64K-word boundary and the last code-space below it.

- If you specify COMPACT, BINSERV checks each succeeding code-space block to determine whether it will fit into either current gap. When a code-space block that fits in a gap is found, BINSERV allocates it in the gap, thereby compressing both the target file and the object program. Figure 10-1 illustrates this process.

Example

```
?COMPACT
```

Figure 10-1. The Effect of the COMPACT Directive



CONSULT Compiler Directive

The CONSULT directive declares external procedures.

When you declare a procedure name in a CONSULT directive, FORTRAN checks the procedure definition and generates an appropriate calling sequence for calls to the procedure that occur in the compilation.

```
CONSULT { consult-item                                  }
        { (consult-item [, consult-item]...)            }
```

consult-item

is a Guardian file name or DEFINE name known at compile time, optionally followed by the name of one or more procedures contained in the referenced file. The procedure names, if present, are separated by commas and contained in parentheses. The syntax of a consult-item is:

```
{ file-name  }
{ define-name }  [( proc-name [, proc-name ]... )]
```

Compile-time defaults are supplied for any missing qualifiers. Each *consult-item* must be an object file created by the Binder. It must contain a Binder region. The procedure bodies in this file can be empty because FORTRAN needs only the procedure names and attributes, along with the attributes of their formal parameters. For more details, see Considerations. For information on the Binder region, see the *Binder Manual*.

Considerations

- The CONSULT directive can appear on the FORTRAN command line following the semicolon after the object file name, or anywhere in the source input file. A compilation can have any number of CONSULT directives.
- If the CONSULT directive names a file without listing procedure names, the FORTRAN compiler looks up all the procedures in that file, and records their descriptions. If the CONSULT directive has a list of procedure names, the compiler looks up only those procedures and records their descriptions.
- If a procedure is named in the directive but not found in the file, the compiler issues a warning message and ignores the procedure, but continues processing the remainder of the directive.
- If two or more procedures with the same name are found (in different files), the compiler uses the first one and ignores the others. The compiler does not report a warning.
- FORTRAN ignores a procedure in a file specified in a CONSULT directive if the procedure:
  - Is a main program.
  - Has a name that is not a legal FORTRAN name.
  - Is a C procedure with lowercase letters in its name.
  - Is a C procedure with a variable number of parameters.
  - Is a C procedure compiled with the OLDCALLS pragma.
  - Was compiled from a source language other than C, COBOL85, FORTRAN, Pascal, or TAL.

- If a *consult-item* names a procedure that is already defined or referenced as a FORTRAN subprogram, the source program's description overrides the description in the file named in the CONSULT directive. If a *consult-item* does not include a list of procedure names, the compiler ignores the procedure without displaying a message. If the procedure is named explicitly in the *consult-directive*, the compiler issues an error message and ignores the procedure.
- You can use CONSULT and GUARDIAN directives in the same compilation. The compiler processes them in the order in which it reads them, and does not give priority to one or the other.
- If you want the FORTRAN compiler to consult files named in LIBRARY and SEARCH directives for calling sequences, you must specify those files in CONSULT directives, as well as in LIBRARY and SEARCH directives.
- One way to create the proper CONSULT *consult-item* is to start with a TAL source file that contains the EXTERNAL declarations of a set of procedures. Use an editor to replace each EXTERNAL with BEGIN END, and compile the modified source file with TAL. The resulting object file can be used with the CONSULT directive and is minimal in size. You can also use an object file that contains the actual procedures.
- Calling a procedure whose description is obtained by a CONSULT directive is the same as calling a procedure whose description is obtained by a GUARDIAN directive. Write a CALL statement or a function reference in the standard way, but with these differences:
  - If the procedure is typed (can be called as a function), it must be called as a function in FORTRAN.
  - If the procedure is VARIABLE or EXTENSIBLE, you can omit parameters in the same way as in TAL; that is, with successive commas, or by omitting unused trailing parameters and their commas.
- Two difficulties can arise when you use CONSULT directives along with SEARCH directives:
  - The CONSULT directive does not guarantee that the procedures will be bound. If you want a particular procedure bound in, you must have a SEARCH directive that names an object file containing that procedure. If you don't want a procedure bound in, you must ensure that no file named in any SEARCH directive has a procedure of that name.
  - When you want a procedure bound in, be sure the file names appear in the same order in the SEARCH and CONSULT directives. Otherwise, the compiler might obtain a procedure's description from one file, but the Binder might get a different procedure with the same name from another file.

  These difficulties can occur because the SEARCH directive allows you to specify file names but not individual procedures within those files. If you need more control over the contents of the object file, use the Binder program after all of your compilations have completed to arrange the object file exactly as desired.

- For information on calling procedures not written in FORTRAN, see Section 13, Mixed-Language Programming.

Example

```
?CONSULT mylib
```

CROSSREF Compiler Directive

The CROSSREF directive instructs the SYMSERV process to generate cross-reference information for selected identifier classes.

```
[NO]CROSSREF [identifier-class
             (identifier-class [, identifier-class]...)]
```

identifier-class

is one or more of the following keywords:

Keyword	Meaning
BLOCKS	COMMON blocks
BLOCKDATAS	BLOCK DATA subprograms
CONSTANTS	PARAMETER-named constants
DUMMYPROCS	Dummy procedures
FMTLABELS	FORMAT statement labels
FUNCTIONS	FUNCTION subprograms
GENERATE	See Considerations
INLINES	Intrinsic functions
LITERALS	Unnamed constants
PROCEDURES	SUBROUTINE subprograms
PROGLABELS	Executable statement labels
STMTFUNCS	Statement functions
UNREF	Unreferenced identifiers
VARIABLES	Variables, arrays, records

Considerations

- Specify the CROSSREF directive either on the FORTRAN command line or in the source file preceding the first FORTRAN statement, or between the END line of one program unit and the first FORTRAN statement of the next program unit.
- If you omit the argument list, the CROSSREF directive lists information for all classes listed in the syntax diagram except LITERALS and UNREF.

- The effect of the CROSSREF directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.
- Using GENERATE

  The compiler normally lists requested cross-reference tables at the end of the compilation. If you specify GENERATE, the compiler lists the requested cross-reference tables at the end of the current program unit, and discards the accumulated references up to that point. At the end of the compilation, the compiler lists the cross-reference information since the last GENERATE listing.
- Using NOCROSSREF

  The NOCROSSREF directive prevents generation of references. It takes effect at the start of the next program unit.

  If you specify a list of classes with the NOCROSSREF directive, the compiler deletes the listed classes from the listing. The compiler does not produce a listing unless a previous CROSSREF directive is in effect.

Example

The following example generates a cross-reference list at the end of the current program unit. The list contains all identifiers except unreferenced variables, unnamed constants, and intrinsic functions.

```
?CROSSREF, NOCROSSREF INLINES, CROSSREF GENERATE
```

DATAPAGES Compiler Directive

The DATAPAGES directive specifies the number of virtual memory pages to allocate for data storage.

```
DATAPAGES number
```

number

is a number in the range 0 through 64.

The default value is equal to the size of the control table plus four pages.

Considerations

- If ENV COMMON is in effect, FORTRAN always allocates 64 data pages, regardless of the value of number.
- If you do not include this directive in the source file, FORTRAN estimates the number of pages to use.

- If you include the directive, Binder allocates precisely the number of pages you specify whether or not that number is sufficient, and displays a warning message if the number you specify is less than the compiler's estimate.

Example

```
?DATAPAGES 16
```

ENDIF Compiler Directive

The ENDIF directive terminates the effect of a preceding IF or IFNOT directive that specifies the same toggle number.

```
ENDIF toggle
```

toggle

is a number in the range 1 through 15.

Considerations

- Write the ENDIF directive as the only item on a directive line.
- Use the IF, IFNOT, and ENDIF directives with the SETTOG and RESETTOG directives to control conditional compilation.
- Note that IFNOT is not equivalent to ELSE.

Example

In the following example, if you use a SETTOG 2 directive, code A is compiled; if you use a SETTOG 3 directive, code B is compiled.

```
?IF 2
   code A
?ENDIF 2

?IF 3
   code B
?ENDIF 3
```

ENV Compiler Directive

The ENV directive determines whether your program uses the C-series or D-series FORTRAN run-time library.

The default value is ENV OLD.

```
ENV { OLD    }
    { COMMON }
```

`OLD`

specifies that this program use the C-series FORTRAN run-time library.

`COMMON`

specifies that this000 program use the D-series FORTRAN run-time library.

If you specify ENV OLD, FORTRAN:

- Generates an object file that contains C-series data blocks that are compatible with the C20 release of the FORTRAN compiler.
- Processes all calls to its run-time environment using FORTRAN run-time library routines and subsequent calls to Guardian system procedures.

If you specify ENV COMMON, FORTRAN:

- Generates an object file that contains D-series data blocks that are defined by the Common Run-Time Environment (CRE).
- Processes calls to its run-time environment using FORTRAN run-time library routines, and subsequent calls to CRE routines and Guardian system procedures.
- Depends on the CRE to:
  - Monitor the backup process of a NonStop process pair
  - Handle traps
  - Provide the math functions required by FORTRAN programs
  - Optionally perform I/O operations for unit 5 and unit 6.

Using ENV COMMON

FORTRAN programs that specify ENV COMMON gain the following benefits:

- You can mix object modules compiled by the C, COBOL85, FORTRAN, Pascal, and TAL compilers more easily. There are restrictions on the operations that each of the language modules can perform, but fewer than when you combine programs that do not specify ENV COMMON.

- The run-time environment provides more consistent and complete run-time error handling for programs that specify ENV COMMON.
- You can specify an eight-character volume name when accessing information over a network.
- FORTRAN supports the NONSTOP and HIGHPIN directives. If ENV OLD is in effect, FORTRAN prints a warning if it encounters a NONSTOP or HIGHPIN directive and ignores the directive.
- FORTRAN supports the EXECUTION-LOG, INSPECT, NONSTOP, and SWITCH-nn TACL PARAMs. For more information about run-time PARAMs, see Section 11, Running and Debugging Programs.

Considerations

- The ENV directive must appear on the FORTRAN command line after the semicolon that follows the object file name, or in the source input file before the first FORTRAN source statement.

  If you specify more than one ENV directive in a compilation, FORTRAN uses the first one you specify and reports a warning message for each subsequent ENV directive that appears before the first FORTRAN statement.

  FORTRAN reports an error if it encounters an ENV directive after the first FORTRAN statement and the ENV directive specifies a different environment than the first ENV directive in your compilation.
- Regardless of whether you specify ENV OLD or ENV COMMON, a FORTRAN subprogram in a user library or in the system library can manipulate only its local data and data items passed to it in its dummy arguments. It can return data only through its dummy arguments and, if it is a function subprogram, as the value returned by the function. It cannot directly access data items in, or equivalenced to, items declared in COMMON, DATA, or SAVE areas, although such items can be passed to the library routine as actual arguments.

  A routine in a library cannot reference items allocated in the extended data segment.

  A library routine or utility routine can execute FORTRAN I/O statements only if the unit number specified in the I/O statement is either specified in a UNIT directive or as a constant value in an I/O statement in a program unit in the program's user code area. You can pass the unit number to the library routine as an actual argument or you can establish the unit number by programmatic convention.

Programs that specify ENV COMMON might be affected as follows:

- You must specify ENV COMMON either for all the FORTRAN programs or for none of the FORTRAN programs that you bind together, including programs bound into your object file as a result of SEARCH directives. You cannot mix FORTRAN programs, subprograms, or functions if some of the program units are compiled with ENV COMMON and others with ENV OLD.

If you bind FORTRAN modules with modules created by other compilers, all the modules that you bind together must specify or default to ENV COMMON, ENV NEUTRAL (TAL routines only), ENV EMBEDDED (Pascal routines only), or ENV LIBSPACE (Pascal routines only). For more information, see the *Binder Manual*.

- Because the library routines in the common environment use extended addresses for all data references, you might notice a small performance degradation when you run your program. The actual performance degradation depends on the percentage of time your program spends executing its own code, compared to the time it spends executing run-time library code.
- FORTRAN programs can open unit connections only to the standard input and standard output files. Although a FORTRAN program cannot open a unit connection to the standard log file, messages written by FORTRAN PAUSE statements, STOP statements, and diagnostic messages from run-time routines are written to the standard log file.

  I/O operations to unit 5 and unit 6 might be processed as they are in C-series systems or might use file sharing routines that apply to the standard files, depending on the values of the file's attributes. For more information about sharing standard files, see the OPEN Statement on page 7-70 and Using ENV COMMON on page 13-26.

  Control blocks and memory buffers are larger than and are located in different locations than programs compiled with ENV OLD. If your program depends on the location of control blocks and buffers, you will need to change your program. You should consider changing your program such that it is not dependent on the size or location of control blocks and memory buffers.
- The size or contents of the following buffers are different depending on whether you specify ENV OLD or ENV COMMON:
  - The array returned in the SOURCE specifier of a READ statement that reads from $RECEIVE
  - The CPLIST parameter to a CHECKPOINT statement and to Saved Message Utility (SMU) routines
- You cannot use the FORTRANCOMPLETION or FORTRANSPOOLSTART routines. Use the FORTRAN_COMPLETION_ routine instead of FORTRANCOMPLETION. Use the FORTRAN_SPOOL_OPEN_ routine instead of the combination of the FORTRANSPOOLSTART routine and the OPEN statement that precedes it. For more information, see Section 15, Utility Routines.
- Run-time diagnostic messages have a different format. If you have a program that depends on the format of run-time diagnostic messages reported by FORTRAN, you must change the program.

Example

In the following example, toggle 2 is reset if ENV OLD is specified; toggle 2 is set if ENV COMMON is specified:

```
?IFNOT 2
?ENV OLD
?ENDIF 2

?IF 2
?ENV COMMON
?ENDIF 2
...

?IFNOT 2
   CALL FORTRANCOMPLETION( ... )
?ENDIF 2

?IF 2
   CALL FORTRAN_COMPLETION_( ... )
?ENDIF 2
```

ERRORFILE Compiler Directive

The ERRORFILE directive saves the error messages from a compilation in a disk file. The disk file can be used with the FIXERRS TACL command to quickly locate and correct the errors in your source code.

```
ERRORFILE file-name
```

file-name

is the name of the file to receive the error messages. It must be either a Guardian file name or a DEFINE name known at compile time. Compile-time defaults are supplied for any missing qualifiers. The file name must be specified. There is no default file name.

Considerations

- The ERRORFILE directive must appear either on the FORTRAN compiler command line after the semicolon following the object file name, or in the source input file preceding the first FORTRAN statement. If you include an ERRORFILE directive after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- If you supply two or more properly placed ERRORFILE directives, the compiler uses the first one and issues a warning message for each of the others.
- If the source input file is anything other than an EDIT format file, the compiler issues a warning message and ignores the ERRORFILE directive.
- The ERRORFILE directive causes the FORTRAN compiler to log all errors and warnings to the specified file, which it creates as an entry-sequenced disk file with file code 106.
- If the file already exists, and it is an entry-sequenced disk file with file code 106, the compiler purges it so the file does not accumulate messages from multiple compilations. If the compiler is unable to purge the existing file, or if the file is anything other than an entry-sequenced disk file with file code 106, the compiler issues a warning message and proceeds as if the ERRORFILE directive were not present.
- The compiler creates the new file only if any error and warning messages are issued during the compilation. The trailer page at the end of the compiler listing states one of the following:

  n messages written to error file *file-name*

  Error file *file-name* was not created

- Using the File Produced by ERRORFILE

  The TACL command FIXERRS runs the TEDIT text editor with one window showing an error message and the other window showing a portion of the source file surrounding the site of the error. After the compilation, enter the TACL command

  ```
  1> FIXERRS file-name
  ```

  or

  ```
  1> FIXERRS file-name ; commands
  ```

  File-name is the name of the error file specified in the ERRORFILE compiler directive, and *commands* is a sequence of one or more TEDIT commands.

  FIXERRS runs TEDIT, which obeys the *commands*, if any, and then gets the first message in the error file. TEDIT displays the text of the error message in the top part of the screen, and the source text surrounding the error in the remainder of the screen. Correct the source text; then use the TEDIT macros NEXTERR and

PREVERR to display other error messages and their source text in the same manner.

You can assign the NEXTERR and PREVERR macros to function keys instead of entering them on the TEDIT command line. For example, in the preceding FIXERRS command, the *commands* part could be a TEDIT USE command that references a TEDIT profile that assigns NEXTERR and PREVERR to function keys.

NEXTERR and PREVERR are distributed as part of TACL; you can find them in the TACL directory named UTILS:FIXERRS. Normally, you don't need to specify a name qualifier or any HOME or USE command to use these macros.

Example

```
?ERRORFILE errfile
```

ERRORS Compiler Directive

The ERRORS directive sets the maximum number of errors for a compilation. When the total number of error messages issued during the compilation (warning messages are not counted) exceeds the value you specify in the directive, the compiler stops without processing the remainder of the source input.

The default is ERRORS 100.

```
ERRORFILE number
```

`number`

is an unsigned integer in the range 0 through 32767.

Considerations

- The ERRORS directive must appear on the FORTRAN command line following the semicolon after the object file name, or in the source input file before the first FORTRAN source statement. If you specify an ERRORS directive after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- If you supply two or more properly placed ERRORS directives, the compiler uses the first one and issues a warning message for each of the others.
- The ERRORS directive is useful when you suspect something is wrong with your source input file. You could specify ERRORS 10, for example, and not waste time trying to compile a bad input file. Limiting the errors can be useful when you are converting a FORTRAN source program from another system.

Example

```
?ERRORS 250
```

EXTENDCOMMON Compiler Directive

The EXTENDCOMMON directive instructs the compiler to use indexed indirect addressing to access simple variables in common blocks in the user data segment.

This method of addressing saves primary global storage but accessing the data in common blocks is slower than if you do not specify EXTENDCOMMON.

The default value is NOEXTENDCOMMON.

```
[NO]EXTENDCOMMON
```

Considerations

- EXTENDCOMMON has no effect on arrays, simple variables more than two words in length, or variables in the extended data segment.
- Normally, FORTRAN allocates one pointer in primary global storage for each entity in common in secondary global storage. If you specify the EXTENDCOMMON directive, FORTRAN allocates only one pointer for the first entity in each common block, one pointer for each array in each common block, and one pointer for all scalars in each common block. The compiler locates entities that follow by indexing.
- Specify the EXTENDCOMMON directive either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement.
- For additional information, see Section 12, Memory Organization.

Example

```
?EXTENDCOMMON
```

EXTENDEDREF Compiler Directive

The EXTENDEDREF directive tells FORTRAN to generate code that uses doubleword addresses for parameters in CALL statements or function references. Programs that use extended data space need doubleword addresses to reference data items stored in extended memory.

The default value is NOEXTENDEDREF, unless you use the LARGECOMMON or

LARGEDATA directives.

```
[NO]EXTENDEDREF
```

Considerations

- Use of EXTENDEDREF Directive

  Use the EXTENDEDREF directive when you compile program units that do not include LARGECOMMON or LARGEDATA directives but which you plan to combine with separately compiled program units that do include LARGECOMMON or LARGEDATA directives.

  You do not have to use the EXTENDEDREF directive in programs that include the LARGECOMMON or LARGEDATA directives. When you use the LARGECOMMON or LARGEDATA directives, FORTRAN automatically compiles your program as if you included an EXTENDEDREF directive.

  You must specify EXTENDEDREF either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement.

  The NOEXTENDEDREF form of the directive tells FORTRAN to use word addressing.

- Use With Separately Compiled FORTRAN Program Units

  You cannot bind FORTRAN program units compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA with FORTRAN program units compiled with NOEXTENDEDREF.

- Use With Guardian Procedures or Utility Routines

  If you call Guardian procedures or FORTRAN utility routines from programs compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA, you must declare the called procedures in GUARDIAN directives.

  Some C-series Guardian procedures use word-addressed parameters. You cannot use a formal parameter or data item stored in extended memory as a passby-reference argument if the dummy argument is a word-addressed item.

  For additional details about using Guardian procedures and FORTRAN utility routines, see Section 13, Mixed-Language Programming.

- Use With COBOL85 Program Units

  You can call a FORTRAN program unit from a COBOL85 program unit whether or not the FORTRAN program unit was compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA. When the COBOL85 compiler processes an ENTER statement, it automatically checks the object file that contains the compiled program unit to determine whether to generate a calling sequence with word or doubleword addresses.

You can call a COBOL85 program unit from a FORTRAN program unit whether or not the FORTRAN program unit uses EXTENDEDREF, LARGECOMMON, or LARGEDATA, but you must code the COBOL85 program unit to specify the addressing mode. The COBOL85 program unit's LINKAGE SECTION must specify "ACCESS MODE IS EXTENDED-STORAGE" if and only if it will be called by a program unit compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA.

Examples

FORTRAN compiles your program with doubleword argument addressing if you specify the following directive:

```
?EXTENDEDREF
```

FORTRAN compiles your program with word argument addressing if you specify the following directive:

```
?NOEXTENDEDREF
```

FIXUP Compiler Directive

The NOFIXUP directive instructs the compiler to omit some of the steps required to make an object file runnable. Using this directive when compiling object files that serve as input to a subsequent Binder run reduces the time required to do a compilation by a few percent.

The default value is FIXUP.

```
[NO]FIXUP
```

Considerations

The FIXUP directive must appear either after the semicolon following the object file name on the TACL command line that runs the FORTRAN compiler or in the source file preceding the first FORTRAN statement.

Example

```
?NOFIXUP
```

FMAP Compiler Directive

The FMAP directive causes the compiler to include a file map in the compiler listing just before the Binder load map. The file map consists of one line for each source input file used during the compilation, showing the file ordinal, the file's fully qualified name, and the date and time the file was last modified.

The default is NOFMAP.

```
[NO]FMAP
```

Considerations

- The FMAP directive must appear on the FORTRAN command line following the semicolon after the object file name, or in the source input file before the first FORTRAN source statement. If you specify the FMAP directive after the first FORTRAN statement, the compiler issues an error message and ignores it.
- If you specify two or more properly placed FMAP directives, the compiler uses the first one and issues a warning message for each of the others.
- The effect of the FMAP directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.
- The file map is most useful when you use DEFINEs for source input file names, because the file map identifies the files that the compiler actually read. Even if you do not use DEFINEs, the file map provides a list of all the source files used in the compilation. Knowing which source files were used can be helpful when you want to be certain you included all the source files necessary to compile the program. FMAP can also be useful if you want to know exactly which versions of the source files FORTRAN used for a particular compilation of a program.

Example

```
?FMAP
```

GUARDIAN Compiler Directive

The GUARDIAN directive specifies the names of Guardian system procedures and of FORTRAN utility routines that your program calls.

If you specify a procedure name in a GUARDIAN directive, FORTRAN checks the procedure definition and generates an appropriate TAL calling sequence for calls your program makes to the procedure.

```
GUARDIAN { proc-name                              }
         { (proc-name [, proc-name]...)           }
```

proc-name

is the name of a Guardian procedure or a FORTRAN utility routine.

If you specify a *proc-name* that is not the name of a Guardian procedure or a FORTRAN utility routine, FORTRAN issues a warning message and generates the normal FORTRAN calling sequence for calls to the procedure.

Considerations

- Use of the GUARDIAN directive

  You must include the GUARDIAN directive that declares a Guardian procedure or a utility routine before the first FORTRAN statement which calls that procedure.

  If you compile any portion of a program with the LARGECOMMON, EXTENDEDREF, or LARGEDATA directives, you must use the GUARDIAN directive to declare all Guardian procedures and FORTRAN utility routines that you call from the program.

  proc-name has the scope of an executable program. If you attempt to redefine it, FORTRAN issues an error message.

- To use the most recent version of a Guardian procedure, use the CONSULT directive instead of the GUARDIAN directive, and specify $SYSTEM.SYSTEM.COBOLEX0.

- Declaring Typed Procedures (functions)

  If you use the GUARDIAN directive to declare a Guardian routine or a utility routine that can be called as a function, the FORTRAN data type returned by the function corresponds to the data type specified by the TAL routine. FORTRAN's normal implicit typing rules do not apply and you do not need to include an explicit type declaration for the function.

- Using Typed Procedures (Functions)

  Unlike TAL, which allows a CALL statement to invoke typed procedures, and generates object code that discards the value returned by the typed procedure,

FORTRAN requires that you invoke a procedure that can be called as a function only by a function reference, not with a CALL statement.

- Value and Reference Parameters

  HP FORTRAN released before B-series systems required that value parameters in calls to Guardian routines be enclosed in backslashes. This convention is no longer necessary. If you add a GUARDIAN directive to a FORTRAN program that still uses backslashes around a value parameter, the compiler issues a warning message.

- VARIABLE and EXTENSIBLE Procedures

  When a VARIABLE or EXTENSIBLE TAL procedure is not declared with a GUARDIAN directive, you must write omitted parameters as \0\, and you must include the parameter mask as one or more additional parameters.

  When you declare such a procedure with the GUARDIAN directive, you indicate omitted parameters with consecutive commas and by omitting commas after the last parameter you specify. You must not pass the parameter mask in the arguments to the called procedure.

Examples

This directive declares the procedure FILEINFO:

```
?GUARDIAN FILEINFO
```

This directive declares the SMU routine GETSTARTUPTEXT:

```
?GUARDIAN GETSTARTUPTEXT
```

This directive declares the procedures FILEINFO, FILERECINFO, and CONTROL:

```
?GUARDIAN (FILEINFO, FILERECINFO, CONTROL)
```

HIGHBUFFER Compiler Directive

The HIGHBUFFER directive specifies the number of words to allocate for run-time data in the upper half of the user data segment.

```
HIGHBUFFER size
```

size

is a number in the range of 1 through 16,383.

If you specify ENV OLD or do not specify an ENV directive, HIGHBUFFER specifies the size of the #HIGHBUF data block, which is allocated in the upper half of the user data segment. If you do not include a HIGHBUFFER directive in the source file, the compiler does not allocate a #HIGHBUFFER data block. All buffered data will be allocated in the #LOWBUF data block.

If you specify ENV COMMON, HIGHBUFFER specifies the size of the #CRE_HEAP data block. #CRE_HEAP contains the private data used by the CRE. If you do not specify HIGHBUFFER, FORTRAN allocates 1,024 words in #CRE_HEAP. If you specify HIGHBUFFER in a program that specifies ENV COMMON, always specify a minimum of 1,024 bytes for *size*.

Considerations

- Your program requires more memory for HIGHBUFFER if you compile your program with ENV COMMON in effect than it does if you compile with ENV OLD in effect.
- The layout of the memory allocated when you specify ENV COMMON is different than the layout if you specify ENV OLD or do not specify an ENV directive.
- For additional information on buffer pools, see Section 12, Memory Organization.

Example

```
?HIGHBUFFER 1066
```

HIGHCOMMON Compiler Directive

The HIGHCOMMON directive allocates common storage in upper data memory for specified common blocks.

```
HIGHCOMMON [block-name
           (block-name  [,  block-name]. . .)]
```

block-name

is a common block name.

Considerations

- The HIGHCOMMON directive must appear before the END statement of the first subprogram that declares the block specified in the directive.
- If you omit the HIGHCOMMON directive, the compiler allocates all COMMON blocks in the lower half of the user data segment or in the extended data segment depending on the LARGECOMMON directive.
- If the HIGHCOMMON directive includes a list of block names, the compiler allocates those blocks in the upper half of the user data segment, and allocates all other blocks in the lower half, or in the extended data segment, depending on the LARGECOMMON directive.
- If you do not specify any common block names for the HIGHCOMMON directive, the compiler allocates all word-addressed blocks in the upper half of the user data

segment, and all byte-addressed blocks in the lower half, except for blocks explicitly specified in LARGECOMMON directives.

- You can use more than one HIGHCOMMON directive in a compilation. The effect is the same as if you had concatenated the common block names in a single directive. As a result, if you include one HIGHCOMMON directive without a block name list and one with, the compiler ignores the directive without a block name list.
- You cannot explicitly mention a byte-addressed block in a HIGHCOMMON directive. A block is byte-addressed if it contains a RECORD or any type CHARACTER data.
- You cannot declare the same block name in a HIGHCOMMON and in a LARGECOMMON directive.
- If your program contains both a HIGHCOMMON and LARGECOMMON directive, one of the directives must have a block name list.

Example

```
?HIGHCOMMON (employee, salary, grade)
```

HIGHCONTROL Compiler Directive

If you compile your program with ENV OLD in effect, the HIGHCONTROL directive instructs Binder to allocate I/O control blocks in the upper half of the user data segment. If you do not specify HIGHCONTROL, the I/O control block resides in secondary global storage below the stack.

If you compile your program with ENV COMMON in effect, the HIGHCONTROL directive instructs Binder to allocate the special data block #MCB in the upper half of the user data segment. If you do not specify HIGHCONTROL, the special data block #MCB resides in the lower half of the user data segment.

The default value is NOHIGHCONTROL.

```
[NO]HIGHCONTROL
```

Example

```
?HIGHCONTROL
```

HIGHPIN Compiler Directive

The HIGHPIN directive specifies whether your program can run at a high PIN.

The default value is NOHIGHPIN.

```
[NO]HIGHPIN
```

Considerations

- The HIGHPIN directive must appear on the FORTRAN command line after the semicolon that follows the object file name, or in the source input file before the first FORTRAN source statement.

  If you specify more than one HIGHPIN directive in a compilation, FORTRAN uses the first one you specify and reports a warning for each subsequent HIGHPIN directive that appears before the first FORTRAN statement.

  FORTRAN reports an error if you specify a HIGHPIN directive after the first FORTRAN statement.

- You can specify HIGHPIN only if you have previously specified ENV COMMON in the same compilation.

- If you use Binder to bind multiple object files together, Binder sets the HIGHPIN attribute in the target file only if all the object files bound into the new object file have the HIGHPIN attribute set. You can use the Binder SET command to establish explicitly the value of the HIGHPIN attribute:

  ```
  SET HIGHPIN { ON  }
              { OFF }
  ```

  You can change the value of the HIGHPIN attribute in an object file using the CHANGE Binder command:

  ```
  CHANGE HIGHPIN { ON  } IN object-file
                 { OFF }
  ```

- Your program runs at a high PIN only if all the following are true:
  - The HIGHPIN attribute is set in the object file.
  - The HIGHPIN attribute is set in the user library file, if your application has a user library.
  - Your process's creator—usually TACL—specifies that your process can run at a high PIN. Processes started from TACL run at a high PIN if the TACL variable

#HIGHPIN is on. You can test the value of #HIGHPIN by entering its name on the TACL command line:

```
1> #HIGHPIN
```

TACL responds with either “YES” or “NO”.

You can change the value of the TACL #HIGHPIN variable by entering the following:

```
1> SET VARIABLE #HIGHPIN { ON  }
                         { OFF }
```

You might not want all processes that you start from TACL to run at a high PIN. Instead of setting #HIGHPIN ON, you can enable high PINs only for the current process by specifying the HIGHPIN run-option:

```
RUN MYPROG / IN file1, OUT file2,... HIGHPIN ... /
```

- The processor in which your process runs is configured to support high PINs and has an available PIN that is greater than 255.

- NOHIGH PIN is the default value for this directive. If you specify HIGHPIN, you might need to make changes in your program. For information about converting processes to run at high PINs, see the *Guardian Application Conversion Guide*.

Example

```
?HIGHPIN
```

HIGHREQ Compiler Directive

The HIGHREQ directive specifies whether your process can be opened as a server by requester processes that run at a PIN that is greater than 255.

The default value is NOHIGHREQ.

```
[NO]HIGHREQ[x]
```

x

is zero or more alphabetic letters or digits. For example, the following directives have the same effect:

```
?HIGHREQ
?HIGHREQS
?HIGHREQUESTORS
?HIGHREQUESTER
?HIGHREQ91
```

Considerations

- The HIGHREQ directive must appear on the FORTRAN command line after the semicolon that follows the object file name, or in the source input file before the first FORTRAN source statement.

  If you specify more than one HIGHREQ directive in a compilation, FORTRAN uses the first one you specify and reports a warning message for each subsequent HIGHREQ directive that appears before the first FORTRAN statement.

  FORTRAN reports an error if you specify a HIGHREQ directive after the first FORTRAN statement.

- The HIGHREQ directive is meaningful only in an object file that includes a main procedure.

- The standalone Binder sets the HIGHREQUESTERS attribute in its target object file equal to the setting of the HIGHREQUESTERS attribute in the object file that contains the main procedure. You can use Binder to explicitly specify the value of the HIGHREQUESTERS attribute:

```
SET HIGHREQUESTERS { ON  }
                   { OFF }
```

Note. The Binder syntax for HIGHREQ requires that you enter the full attribute name, as in:

```
SET HIGHREQUESTERS ON
```

You cannot specify:

```
SET HIGHREQ ON
```

- Your program accepts requests from processes running at high PINs:
  - If your object file specifies HIGHREQ.
  - If your program opens $RECEIVE by explicitly calling the FILE_OPEN_ system procedure, rather than the OPEN system procedure.

  - If you specify ENV COMMON, and do not open $RECEIVE by explicitly calling the OPEN system procedure. If your program specifies ENV COMMON, FORTRAN opens $RECEIVE with FILE_OPEN_, regardless of the value of the HIGHREQ directive.

- If a process running at a high PIN attempts to open your process and your process is running with NOHIGHREQ, the file system returns error code 560.
- If you compile your program with both ENV OLD and HIGHREQ in effect, and your program reads from $RECEIVE, the SOURCE specifier in the READ statement will contain a CRTPID (creation timestamp process identifier) in SOURCE(5:8). You can use the CRTPID only to compare it with the CRTPID specified in a CPU or NODE status change message, or to compare it to the CRTPID returned to your program when it received an OPEN system message from a requester.

  If you need to use a process identification string in any other context, you must specify ENV COMMON or NOHIGHREQ. If you specify ENV COMMON, the value returned in the SOURCE array is a process handle, rather than a CRTPID. For more information, see Section 14, Interprocess Communication. For more information about CRTPIDs and process handles, see the *Guardian Programmer's Guide*.
- NOHIGHREQ is the default value for this directive. If you specify HIGHREQ, you might need to make changes in your program. For information about converting your process to handle high-PIN requesters, see the *Guardian Application Conversion Guide*.

Example

```
?HIGHREQUESTERS
```

ICODE Compiler Directive

The ICODE directive instructs the compiler to list the symbolic instruction codes it generates for each program unit, following the source listing for that program unit. The effect of the ICODE directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.

The default value is NOICODE.

```
[NO]ICODE
```

Example

```
?LIST, ICODE
```

IF Compiler Directive

The IF directive identifies the beginning of a sequence of source records that FORTRAN compiles only if a specified toggle is set. The compiler continues compiling source records until it encounters either an ENDIF directive that specifies the same toggle number as is specified in the IF directive or until it encounters the end of the source file.

If the toggle specified on the IF directive is reset, FORTRAN skips all subsequent input records until it encounters an ENDIF directive with the same toggle number as is specified in the IF directive.

```
IF toggle
```

toggle

is a number in the range 1 through 15.

Considerations

- Write the IF directive as the last item on a directive line.
- If you have not set *toggle* with a SETTOG directive, the compiler skips the source lines following the IF directive.
- You cannot nest IF directives.
- Once you have initiated skipping of source text using an IF directive, the compiler skips source lines until the next matching ENDIF. Thus, the way to write an if-then-else decision structure is:

```
?IF n
   (statements to be compiled if toggle n is set)
?ENDIF n
?IFNOT n
   (statements to be compiled if toggle n is reset)
?ENDIF n
```

- The compiler prints a “#” character in the listing file to the left of each source line that it skips as a result of an IF directive.

Example

In the following example, if you use a SETTOG 2 directive, code A is compiled; if you use a SETTOG 3 directive, code B is compiled.

```
?IF 2
   code A
?ENDIF 2

?IF 3
   code B
?ENDIF 3
```

IFNOT Compiler Directive

The IFNOT directive identifies the beginning of a sequence of source records that FORTRAN compiles only if a specified toggle is reset. The compiler continues compiling source records until it encounters either an ENDIF directive that specifies the same toggle number as is specified in the IFNOT directive or until the compiler encounters the end of the source file.

If the toggle specified on the IFNOT directive is set, FORTRAN skips all subsequent input records until it encounters an ENDIF directive with the same toggle number as is specified in the IFNOT directive.

```
IFNOT toggle
```

toggle

is a number in the range of 1 through 15.

Considerations

- Write the IFNOT directive as the last item on a directive line.
- You cannot nest IFNOT directives.

- Once you have initiated skipping of source text using an IFNOT directive, source lines are skipped until the next matching ENDIF. Thus, the way to write an if-then-else decision structure is:

```
?IF n
   (statements to be compiled if toggle n is set)
?ENDIF n
?IFNOT n
   (statements to be compiled if toggle n is reset)
?ENDIF n
```

- The compiler prints source lines skipped as a result of an IFNOT directive in the source text listing with a “#” character to the left of the line.

Example

In the following example, the RESETTOG directive instructs the compiler to compile code C. If you use a RESETTOG 2 and RESETTOG 3 directive, FORTRAN compiles both code A and code B.

```
?RESETTOG 3

?IFNOT 2
   code B
?ENDIF 2

?IFNOT 3
   code C
?ENDIF 3
```

INSPECT Compiler Directive

The INSPECT directive establishes the default debugger for the object file. INSPECT selects the Inspect debugger; NOINSPECT selects the Debug debugger.

The default value is NOINSPECT.

```
[NO]INSPECT
```

Considerations

- Specifying the SAVEABEND directive automatically selects the Inspect debugger.
- Make sure Inspect is available on your system if you specify INSPECT.
- For additional information, see Section 11, Running and Debugging Programs.

Example

```
?INSPECT
```

INTEGER Compiler Directive

The INTEGER directive specifies the size of each variable you declare in an INTEGER declaration statement that does not include a size specification. The INTEGER directive also specifies the size of variables whose first letter implicitly designates an integer.

If you omit this directive, all integer entities not otherwise declared are INTEGER*2.

```
{INTEGER*2}
{INTEGER*4}
{INTEGER*8}
```

Considerations

- Use an INTEGER*4 directive for programs that require that FORTRAN allocate a doubleword for variables declared INTEGER or for programs that rely on storage allocation that conforms to the ANSI standard's requirement that all integer, logical, and real variables occupy the same amount of storage space. However, using the INTEGER*4 directive will make your object code larger and your program run slower than if you use INTEGER*2 (that is, word) variables.
- Specify the INTEGER directive either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement.

Example

```
?INTEGER*4
```

LARGECOMMON Compiler Directive

The LARGECOMMON directive tells FORTRAN to allocate space for common blocks in extended memory.

If you specify block names in the LARGECOMMON directive, FORTRAN uses extended memory for each common block you specify in the directive.

If you omit block names from the LARGECOMMON directive, FORTRAN uses extended memory for all common blocks in the compilation that you do not explicitly specify in HIGHCOMMON directives.

```
LARGECOMMON { block-name                              }
            { (block-name [, block-name]...)          }
```

block-name

is the name of a common block.

Considerations

- Use of the LARGECOMMON Directive

  You must specify LARGECOMMON either with the FORTRAN command (after the semicolon following the object file name), or in your source file before the first FORTRAN statement.

  If you specify the LARGECOMMON directive, FORTRAN compiles your program as if you had specified an EXTENDEDREF directive, even if you did not include an EXTENDEDREF directive. (For additional considerations, see the EXTENDEDREF Compiler Directive on page 10-24.)

  You cannot use the LARGECOMMON directive in a compilation that specifies NOEXTENDEDREF.

  If you use both the LARGECOMMON and HIGHCOMMON directives in the same compilation, at least one of the two directives must include one or more block names. You cannot specify the same block name in both a LARGECOMMON and HIGHCOMMON directive.

  Do not use the LARGECOMMON directive in programs that call Guardian procedures such as SEGMENT_ALLOCATE_ to allocate extended data space.

  Your direct handling of extended data space—through calls to Guardian procedures—can interfere with the memory management required for the

LARGECOMMON directive. Use LARGECOMMON directives, not Guardian procedure calls, if your program needs extended data space.

- Executing Programs Compiled With LARGECOMMON

  Accessing variables in normal user data space is faster than accessing variables in extended data space. As a result, a program without LARGECOMMON directives executes faster than an otherwise equivalent program that includes LARGECOMMON directives.

- Common Block Memory Allocation

  Normally, FORTRAN allocates space for common blocks in the lower half of user data segment.

  Use the LARGECOMMON directive to allocate common blocks in extended memory; use the HIGHCOMMON directive to allocate common blocks in the upper half of the user data segment.

Examples

The following directive tells FORTRAN to allocate extended memory for all common blocks in the compilation except those you explicitly list in HIGHCOMMON directives:

```
?LARGECOMMON
```

The following directive tells FORTRAN to allocate extended memory for common block xxx:

```
?LARGECOMMON xxx
```

FORTRAN allocates all other common blocks in the compilation in the user data segment unless you include additional LARGECOMMON directives that specify names of other common blocks. The following directives tell FORTRAN to allocate extended memory for four common blocks—aaa, bbb, ccc, and ddd:

```
?LARGECOMMON (aaa, bbb, ccc)
?LARGECOMMON ddd
```

LARGEDATA Compiler Directive

The LARGEDATA directive causes the compiler to allocate memory space in the object program's extended data segment for local data.

```
LARGEDATA [item                    ]
          [item [, item]...        ]
```

item

is a simple variable name, an unsubscripted array name, an unsubscripted unqualified RECORD name, or an unsigned integer constant.

If *item* is not a constant, FORTRAN allocates space for *item* in the extended data segment, rather than in the user data segment.

If *item* is an integer constant, FORTRAN allocates space in the extended data segment for all local variables having a size of at least *item* bytes. If you omit *item*, FORTRAN interprets this as:

```
?LARGEDATA 256
```

If you specify two or more integer constant *items*, the compiler uses the smallest of their values. If an *item* has a zero value, the compiler does not assign any data objects to the extended data segment.

Considerations

- You can include as many LARGEDATA directives in your program as you like, and you can place them anywhere in the input to the compiler.
  - If you specify the LARGEDATA directive using the implied RUN command for the compiler, or specify it in the source file before the first FORTRAN statement, the directive applies to all the program units in the compilation.
  - If you specify a LARGEDATA directive after the first FORTRAN statement of your program, it applies to the program unit it is specified in.
- Specifying an *item* using a LARGEDATA directive causes the compiler to allocate memory space in the object program's extended data segment for that *item* and any item associated with it in an EQUIVALENCE statement.
- The compiler allocates an *item* that you also declare with a DATA or SAVE statement permanently and statically. It allocates all other *items* you specify in the LARGEDATA directive dynamically on a run-time stack.
- The compiler ignores an *item* specification in a program unit in which the item is known as a subprogram name, an entry point name, a statement function name, a dummy argument name, or a symbolic constant name, or is in a common block, or is equivalenced to a variable that is in a common block.

- Specifying LARGEDATA automatically selects EXTENDEDREF.
- If a LARGEDATA directive appears anywhere in the compiler's input, at least one EXTENDEDREF, LARGECOMMON, or LARGEDATA directive must precede the first FORTRAN statement. You cannot use the LARGEDATA directive in a compilation that uses the NOEXTENDEDREF directive.
- You cannot use the LARGEDATA directive for FORTRAN program units executed from a user library object file or from the system library.
- Do not use the LARGEDATA directive in programs that call Guardian procedures (for example, the SEGMENT_ALLOCATE_ procedure) to allocate extended data space. Your direct handling of extended data space through calls to Guardian procedures can interfere with the memory management required for the LARGEDATA directive. Use LARGEDATA directives, not Guardian procedure calls, if your program needs extended data space.

Program Conversion Considerations

When converting an existing FORTRAN program to run on HP NonStop systems, you should not use a LARGEDATA directive at first. If execution ends with a "stack overflow" error, try adding a LARGEDATA directive with no *items* at the beginning of the source program.

If "stack overflow" still occurs, consider the following:

- You might be executing a recursive procedure for which the termination condition is never met.
- You might need to use SAVE statements to make some local variables static.
- You might need to use the LARGESTACK directive to specify the size of the extended stack area.

If the program runs correctly but is too large or too slow, consider replacing the blank LARGEDATA directive with a set of more specific ones.

Examples

```
?LARGEDATA (names, addresses, birth)

?LARGEDATA 128
```

LARGESTACK Compiler Directive

The LARGESTACK directive specifies the block size to reserve for dynamically allocated variables specified in LARGEDATA directives.

```
LARGESTACK number
```

number

is an unsigned decimal integer that specifies the number of memory pages to allocate for dynamically allocated variables specified in LARGEDATA directives.

Considerations

- The compiler calculates the block size to reserve for dynamically allocated variables specified in LARGEDATA directives as the sum of the block sizes in all the program units in the compilation.

  The LARGESTACK directive overrides the compiler's calculation.

- You might need to use a LARGESTACK directive if your program uses recursion or you use Binder to bind multiple object files together and you want to ensure that the block size for dynamically allocated variables in the new object file created by Binder is large enough for the executable program.
- The LARGESTACK directive must appear on the FORTRAN command line following the semicolon after the object file name, or in the source input file before the first FORTRAN source statement.
- If you specify a LARGESTACK directive after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- If you specify two or more properly placed LARGESTACK directives, the compiler uses the first one, and issues a warning message for each of the others.
- The Binder command SET LARGESTACK can also be used to set the extended memory stack size for an object file.

Example

```
?LARGESTACK 512
```

LIBRARY Compiler Directive

The LIBRARY directive establishes the default user library for the object file. When you run your program, the system consults the user library file specified in the LIBRARY directive for any unsatisfied external procedure references in the object file, before it consults the system library code space.

```
LIBRARY file-name
```

file-name

is a Guardian file name that specifies an object file to use as the default user library file for the object file. It cannot be a DEFINE name. Compile-time defaults are supplied for any missing qualifiers in *file-name*. *file-name* must be a Binder object file.

Considerations

- The LIBRARY directive must appear on the FORTRAN command line following the semicolon after the object file name, or in the source input file before the first FORTRAN source statement.
- If you specify a LIBRARY directive after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- If you specify two or more properly placed LIBRARY directives, the compiler uses the first one, and issues a warning message for each of the others.
- If you want the FORTRAN compiler to consult the user library file for calling sequences, you must specify that user library file in a CONSULT directive and in the LIBRARY directive.
- If the RUN command includes a LIB *file-name* run-option, that file is used instead of the file named in the LIBRARY directive. That is, the run-time specification overrides the compile-time specification.
- You can include TAL subprograms in the user library object file. You can also include subprograms written in FORTRAN, provided they don't:
  - Directly reference data items declared in COMMON, DATA, or SAVE statements, or data items that are declared equivalent to such items.
  - Directly reference data items allocated in the extended data segment as a result of any LARGEDATA directives.

For more information about libraries, see Section 9, Program Compilation and Section 11, Running and Debugging Programs.

Example

```
?LIBRARY mylib
```

LINES Compiler Directive

The LINES directive specifies the number of lines the compiler writes to each page of the listing file.

```
LINES number
```

number

is a number ranging from 10 through 32767. The default value is 60.

Considerations

If you use more than one LINES directive, the new value for *number* takes effect when the directive is scanned.

If the value you supply for *number* lies outside the legal range, FORTRAN displays a warning message and uses either the previous value for *number* if there was one, or the default value.

Example

```
?LINES 55
```

LIST Compiler Directive

The LIST directive controls the listing of source lines and enables the CODE, CROSSREF, ICODE, FMAP, LMAP, MAP, and PAGE directives.

The default value is LIST.

```
[NO]LIST
```

Example

```
?NOLIST
```

LMAP Compiler Directive

The LMAP directive instructs BINSERV to pass load-map information to the compiler. The compiler then lists load maps after its identifier map and cross-reference tables.

The default value is LMAP ALPHA.

```
[NO]LMAP [list-option
         (list-option [, list-option]...)]
```

list-option

is any of the following:

ALPHA

specifies maps in alphabetic order.

LOC

specifies maps in order by base address.

XREF

specifies a cross-reference listing of all entry points and data blocks in the object file.

*

specifies ALPHA, LOC, and object-file cross-references of entry points and data blocks.

Considerations

A data-block map follows the entry point table. This map lists all common blocks and compiler-generated special data blocks identified by names that contain a “#” character. Data-block entries give the base and limit of the block. If the limit field is blank, the data block is empty.

The effect of the LMAP directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.

Example

```
?LMAP *
```

LOGICAL Compiler Directive

The LOGICAL directive specifies the size (in bytes) of all subsequent entities in the source file that are declared as type logical.

```
{INTEGER*2}
{INTEGER*4}
```

Considerations

- You cannot include more than one LOGICAL directive in your source program. Specify the LOGICAL directive either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement.
- If you omit this directive, all logical entities are LOGICAL*2.
- Use the LOGICAL*4 directive, along with the INTEGER*4 directive, for programs that rely on storage allocation conforming to the ANSI standard's requirement that all integer, logical, and real variables occupy the same amount of storage space.

Example

```
?LOGICAL*4
```

LOWBUFFER Compiler Directive

The LOWBUFFER directive controls space allocated for the run-time buffer pool in lower data memory.

```
LOWBUFFER size
```

size

is a number in the range 0 through 16383 that specifies the number of words to allocate.

Considerations

- The LOWBUFFER directive is meaningful only if ENV OLD is in effect. If you specify ENV COMMON, the LOWBUFFER directive has no effect on space allocation for your program.
- If you do not include the LOWBUFFER directive in your source file, the compiler allocates 512 words.

- The run-time buffer pool provides space for edit control blocks (for programs that run as NonStop processes), level-3 spooling buffers, saved messages, and $RECEIVE tables. The amount of space specified or assumed by the LOWBUFFER directive is in addition to the amounts specified or assumed by the SAVE and RECEIVE directives.
- For a FORTRAN program running as a NonStop process, the default buffer pool size of 512 words allows you to have up to three EDIT format files open at a time, since each requires 169 words for its edit control block. If you will have four or more EDIT format files open at the same time, you must use the LOWBUFFER directive to increase the buffer space to prevent a file system error 32, “unable to obtain storage pool space,” during execution.

  A program not that does not run as a NonStop process does not use space in this area for EDIT format files because it allocates EDIT format file buffers in a separate extended data segment, inaccessible to the FORTRAN program.
- Each level-3 spooling output file requires a 512-word buffer in either this area or the HIGHBUFFER area, unless a call to FORTRANSPOOLSTART explicitly provides a buffer area for the file.
- For additional information about buffer pools, see Section 12, Memory Organization.

Example

```
?LOWBUFFER 1024
```

MAP Compiler Directive

The MAP directive instructs the compiler to list, following each program unit’s source listing, a table of local identifiers for that program unit. MAP also lists a table of entities in common storage following the last program unit’s listing.

The effect of the MAP directive is suspended, but not cancelled, by the NOLIST and SUPPRESS directives.

The default value is MAP.

```
[NO]MAP
```

Considerations

Specify the MAP directive either with the FORTRAN command (following the semicolon after the object file name) or in the source input file preceding the first FORTRAN statement, or between the END line of one program unit and first FORTRAN statement of the next program unit.

Example

```
?NOMAP
```

NONSTOP Compiler Directive

The NONSTOP directive specifies that you want your program to run as a NonStop process.

The default value is NONONSTOP.

```
[NO]NONSTOP
```

If your program specifies or defaults to ENV OLD:

- FORTRAN reports a warning if it encounters a NONSTOP directive.
- FORTRAN specifies in the object file that your program run as a NonStop process if your program includes either a START BACKUP or a CHECKPOINT statement.
- You cannot disable START BACKUP statements.

If your program specifies ENV COMMON:

- A NONSTOP directive enables your program to run as a NonStop process, even if your program does not include a START BACKUP or a CHECKPOINT statement.
- The Binder specifies that the object file it produces can run as a NonStop process only if the object file from which Binder reads the main procedure can run as a NonStop process.
- When you run your program, FORTRAN does not process START BACKUP statements if you do not specify the NONSTOP directive or if you specify a PARAM NONSTOP OFF TACL command.
- If you specify the NONSTOP directive, the FORTRAN run-time system reports error 257 if you attempt to open an EDIT format file with MODE = 'OUTPUT' or MODE = 'I-O'.

Considerations

The NONSTOP directive must appear on the FORTRAN command line after the semicolon that follows the object file name, or in the source input file before the first FORTRAN source statement.

If the source file contains an ENV directive, the NONSTOP directive must appear after the ENV directive.

If you specify more than one NONSTOP directive in a compilation, FORTRAN uses the first one you specify and reports a warning message for each subsequent NONSTOP directive that appears before the first FORTRAN statement.

FORTRAN reports an error if it encounters a NONSTOP directive after the first FORTRAN statement.

Example

```
?NONSTOP
```

PAGE Compiler Directive

The PAGE directive ejects the current page of the list file, prints the specified character string at the top of the next page, and skips two lines before resuming the listing.

```
PAGE [ " title" ]
```

title

is a character string. FORTRAN interprets two immediately adjacent quotation mark characters within *title* as one quotation mark character when it writes title to the listing file.

Considerations

- The PAGE directive must be the first or only directive on a line.
- The first PAGE directive establishes the title without skipping a page.
- If you do not specify a title, PAGE uses the previous title.
- The PAGE directive does not cause a page eject when NOLIST or SUPPRESS is in effect, but if a new title is specified, it will appear at the top of the next page after printing resumes.
- If title contains more than 60 characters, FORTRAN uses only the first 60 characters.

Examples

```
?PAGE "New Spelling Checker"

?PAGE "Listing for ""First Compilation"""
```

POP Compiler Directive

The POP directive restores the state of a directive that was saved by a previous PUSH directive.

```
POP { directive                    }
    { directive [, directive]...   }
```

directive

is any of the following FORTRAN compiler directives:

BOUNDSCHECK

CODE

ICODE

LIST

MAP

PRINTSYM

WARN

The PUSH and POP directives can be useful in auxiliary source input files that are referenced from other files by SOURCE directives. Within an auxiliary source file, you can save the state of certain compiler directives when you begin compiling statements from the file, and restore the state of those directives when you complete reading from the auxiliary source file, without knowing what the surrounding context was.

Considerations

- You can specify a POP directive on the FORTRAN command line, or anywhere in any source input file, provided that *directive* is permitted there. You can have any number of POP directives in a compilation.
- The compiler maintains a push-down stack with a maximum of 16 elements for each directive that can be pushed and popped. If more than 16 elements have been pushed into a directive's stack, the oldest elements are lost. No message is given when an element is lost.
- If more than 16 elements have been popped from a directive's stack, the POP directive restores the default state for that directive.

Example

In a subroutine for which array subscript bounds checking is particularly important, the source file could have:

```
?PUSH BOUNDSCHECK
?BOUNDSCHECK                < -- At its beginning
.
?POP BOUNDSCHECK            < -- At its end
```

Such a source file can be incorporated into any program with a SOURCE directive without disturbing the bounds checking mode of the program using it.

PRINTSYM Compiler Directive

The PRINTSYM directive causes the compiler to include or omit unreferenced identifiers in MAP listings.

The default is NOPRINTSYM.

```
[NO]PRINTSYM
```

Considerations

- You must specify the PRINTSYM directive on the FORTRAN command line following the semicolon after the object file name, or anywhere in the source input file. You can use any number of PRINTSYM directives in a compilation.
- If you declare a symbolic name while NOPRINTSYM is in effect, but the name is not referenced in any EQUIVALENCE statements or in any executable statements, the name is omitted from the MAP for the scope in which it is declared. A symbolic name declared while PRINTSYM is in effect is included in the MAP, even if it is not otherwise referenced.

Example

```
?PRINTSYM
```

PUSH Compiler Directive

The PUSH directive causes the current state of a compiler directive to be saved in a push-down stack where it can be restored by a later POP directive specifying the same directive. The PUSH directive does not change the state of the subject directive.

```
PUSH { directive                          }
     { directive [, directive]...         }
```

directive

is any of the following FORTRAN compiler directives:

BOUNDSCHECK

CODE

ICODE

LIST

MAP

PRINTSYM

WARN

The PUSH and POP directives can be useful in auxiliary source input files that are referenced from other files by SOURCE directives. Within an auxiliary source file, you can save the state of certain compiler directives when you begin compiling statements from the file, and restore the state of those compiler directives when you complete reading from the auxiliary source file, without knowing what the surrounding context was.

Considerations

- You can specify the PUSH directive on the FORTRAN command line, or anywhere in any source input file, provided that *directive* is permitted there. You can have any number of PUSH directives in a compilation.
- The compiler maintains a push-down stack with a maximum of 16 elements for each directive that can be pushed and popped. If more than 16 elements are pushed into a directive stack, the oldest elements are lost. No message is given when an element is lost.

Example

In a subroutine for which array subscript bounds checking is particularly important, the source file could have:

```
?PUSH BOUNDSCHECK
?BOUNDSCHECK              <-- At its beginning
.
?POP BOUNDSCHECK          <-- At its end
```

Such a source file can be incorporated into any program with a SOURCE directive without disturbing the bounds checking mode of the program using it.

RECEIVE Compiler Directive

The RECEIVE directive enables you to specify values for parameters that control the length of a reply, the number of processes that can open this process, the number of messages that can be posted to this process at any given time, the number of messages that you want resent in the event of a failure, and whether you want to receive system messages.

```
RECEIVE { receive-spec                                  }
        { (receive-spec [, receive-spec]...)            }
```

receive-spec

is one of the following:

MAXREPLY *reply*

reply is an integer in the range 0 through 32767 that specifies the maximum number of bytes you can include when you reply to a message previously received from $RECEIVE. The default value for MAXREPLY is 132 if ENV OLD is in effect, 0 if ENV COMMON is in effect.

OPEN *open*

open is an integer in the range 1 through 255 that specifies the maximum number of processes that can open your process simultaneously (not counting backup opens). If you specify SYSMSG, the value you specify for *open* must include the number of processes that can open your process plus an additional open by the operating system. The default value is 1.

QDEPTH *depth*

depth is an integer in the range 1 through 255 that specifies the maximum number of messages from $RECEIVE that your process can hold at any one

time. Your process cannot receive additional messages until you reply to at least one of the messages already received. The default value is 1.

SYNCDEPTH *sync*

sync is an integer in the range 1 through 255 that specifies the maximum number of replies this process saves for each opener. The default value is 1.

SYSMSG

specifies that FORTRAN return system messages received from $RECEIVE to your program. You cannot queue system messages. If a system message requires a reply—for example, an OPEN system message—the next write to $RECEIVE must be a reply to the system message.

Considerations

- If you specify a RECEIVE directive more than once, FORTRAN uses the last value specified.
- If *open* processes have your process open, FORTRAN returns error 12, “file in use,” to processes that attempt to open your process.
- The record length of the unit receiving system messages through $RECEIVE must be at least 34 characters.
- If you omit the SYSMSG option, the FORTRAN facility handles system messages.
- Table 10-2 lists the C-series and D-series system messages that your process might receive if you specify SYSMSG.

Table 10-2. System Messages (page 1 of 2)

C-Series		D-Series	
Msg No.	**Message Text**	**Msg No.**	**Message Text**
-2	CPU down (MONITORCPUS)	-2	CPU down (MONITORCPUS)
-2	CPU down: named process deletion	-101	Process deletion: CPU down
-3	CPU up	-3	CPU up
-8	Change in status of network node	-100	Remote CPU down
-8	Change in status of network node	-110	Loss of communication with node
-8	Change in status of network node	-111	Establishment of communication with node
-8	Change in status of network node	-113	Remote CPU up
-10	SETTIME	-10	SETTIME
-11	Power ON	-11	Power ON

Table 10-2. System Messages (page 2 of 2)

C-Series		D-Series	
Msg No.	**Message Text**	**Msg No.**	**Message Text**
-12	NEWPROCESSNOWAIT completion	-102	Nowait PROCESS_CREATE_ completion
-20	Break on device	-105	Break on device
-22	Elapsed time timeout	-22	Elapsed time timeout
-30	Process OPEN	-103	Process OPEN
-31	Process CLOSE	-104	Process CLOSE
-32	Process CONTROL	-32	Process CONTROL
-33	Process SETMODE	-33	Process SETMODE

- If you specify a value other than 132 for reply, you must use the same value for the record length of the unit that you use to send replies to $RECEIVE. You can do this with the REC option of an ASSIGN command, or a UNIT compiler directive, or the RECL specifier of an OPEN statement.
- For additional information, see Section 14, Interprocess Communication.

Examples

```
?RECEIVE (OPEN 2, SYNCDEPTH 5, MAXREPLY 132)
?RECEIVE SYSMSG
```

RESETTOG Compiler Directive

The RESETTOG directive resets toggles used to control conditional compilation.

```
RESETTOG [ toggle [, toggle ]... ]
```

toggle

is a number in the range of 1 through 15 that specifies a toggle to reset.

Considerations

- Write the RESETTOG directive as the last item on a directive line.
- If you do not specify any toggles, the RESETTOG directive resets all 15 toggles.
- The compiler initially resets all 15 toggles.
- Use the RESETTOG and SETTOG directives with the IF, IFNOT, and ENDIF directives to control conditional compilation.

Example

```
?RESETTOG 1,3,5
```

RUNNAMED Compiler Directive

The RUNNAMED directive specifies that your program run as a named process.

The default value is NORUNNAMED.

```
[NO]RUNNAMED
```

By specifying the RUNNAMED directive, your process can run at a high PIN and be accessed by processes that have not been converted to use D-series features. See, also HIGHPIN Compiler Directive on page 10-32.

Considerations

- The RUNNAMED directive must appear on the FORTRAN command line after the semicolon that follows the object file name, or in the source input file before the first FORTRAN source statement.

  If you specify more than one RUNNAMED directive in a compilation, FORTRAN uses the first one you specify and reports a warning message for each subsequent RUNNAMED directive that appears before the first FORTRAN statement.

  FORTRAN reports an error if you specify a RUNNAMED directive after the first FORTRAN statement.

- If you use Binder to bind multiple object files together, Binder sets the RUNNAMED attribute to ON in the new object file if any of the object files bound into the new object file specify RUNNAMED ON.

  You can use the Binder SET command to establish explicitly the value of the RUNNAMED attribute:

  ```
  SET RUNNAMED { ON  }
               { OFF }
  ```

- Your program runs as a named process:
  - If the RUNNAMED attribute is set in the object file.
  - If the process that creates your process specifies that your process run as a named process.

  If your process runs as a named process but you do not specify a process name when you run your program, the operating system creates a unique process name for you.

Example

```
?RUNNAMED
```

SAVE Compiler Directive

The SAVE directive specifies the system messages to save during process initialization.

```
SAVE { save-spec                          }
     {(save-spec [, save-spec]...)        }
```

save-spec

is one of the following:

`STARTUP [ stup ]`

stup is an integer in the range 68 through 594 that specifies the number of bytes of buffer space to allocate for the saved startup message. The default value is 594.

`PARAM [ param ]`

param is an integer in the range 4 through 1028 that specifies the number of bytes of buffer space to allocate for PARAM messages. The default value is 1028.

`ASSIGNS [ mess ]`

mess is an integer in the range 1 through 100 that specifies the number of ASSIGN messages for which buffer space is reserved. The default value is 15.

`ALL [ allmsg ]`

allmsg is an integer in the range 1 through 100 that specifies a number of ASSIGN messages for which buffer space is reserved. The default value is 15.

Section 15, Utility Routines, explains how to use the saved information.

Considerations

- Use parentheses if you specify more than one option for the SAVE directive.
- Use the STARTUP parameter to save the command interpreter startup message.
- Use the PARAM parameter to save messages generated by PARAM commands.
- Use the ASSIGNS parameter to save messages generated by ASSIGN commands.

- Use the ALL parameter to save the STARTUP message, PARAM message, and all ASSIGN messages.
- If you omit the SAVE directive, FORTRAN does not save any messages or reserve any buffer space.

Example

```
?SAVE ALL
```

SAVEABEND Compiler Directive

The SAVEABEND directive specifies whether Inspect should automatically create a save file if the program terminates abnormally at run time. Use the SAVEABEND directive only with the INSPECT directive.

The default value is NOSAVEABEND.

```
[NO]SAVEABEND
```

Considerations

- The save file captures the state of your program's data and file status information at the point of failure. You can use Inspect to examine the save file.
- Inspect creates the save file on the same volume as the program file and assigns it a name in the form:

  ZZSA *nnnn*

  where *nnnn* is a random number.
- For additional information. Example, see Section 11, Running and Debugging Programs.

Example

```
?INSPECT, SAVEABEND
```

SEARCH Compiler Directive

The SEARCH directive specifies a list of object files for BINSERV to search for unsatisfied external references at compilation time.

```
SEARCH { file-name                          }
       { (file-name [, file-name]...)       }
```

file-name

is the name of a disk file that contains object code produced by the C, COBOL85, FORTRAN, Pascal, or TAL compiler. *file-name* can be a DEFINE name.

Considerations

- At the end of the compilation, BINSERV searches the files listed in the SEARCH directive in the order specified. Then, BINSERV searches $SYSTEM.SYSTEM.FORTLIB.
- You can extend the SEARCH directive over more than one line. Begin continuation lines with a question mark in column one.
- A SEARCH directive with an empty file list clears the search list.
- The object file produced at the end of the compilation includes copies of the code and data blocks found by the SEARCH directive. If these code and data blocks also contain external references, BINSERV uses the search list to satisfy those external references as well. Thus, the target file contains all required code and data that is available via the search list.
- For additional information, see Section 9, Program Compilation.

Example

```
?SEARCH ( object1, object2, object3 )
```

SECTION Compiler Directive

The SECTION directive assigns a name to a section of a source file for use in SOURCE directive in another source file.

```
SECTION section-name
```

section-name

is a symbolic name of up to 31 characters that can be a combination of A Z, 0 through 9, and the special characters circumflex (^), hyphen (-), and underscore (_). The first character of the name must be a letter.

Considerations

- The *section-name* identifies all source text that follows the SECTION until another SECTION directive or the end of the source file occurs.
- The SECTION directive must be the only directive on the directive line.

Example

The following example includes a file, FUNCTIONS, with a SECTION directive identifies a section called MATHROUTINES. The program called MAIN sources the text from the section MATHROUTINES in the file FUNCTIONS.

```
File: FUNCTIONS.
   .
?SECTION mathroutines
   SUBROUTINE random (a,b,c)
   .
   END
?SECTION buildarray
   SUBROUTINE array (x,y,z)
   .
   END
```

You source in the RANDOM subroutine by including the following directive program:

```
      PROGRAM main
?SOURCE function ( mathroutines )
      .
      END
```

SETTOG Compiler Directive

The SETTOG directive sets toggles that control conditional compilation.

```
SETTOG [ toggle [, toggle ]... ]
```

toggle

is a number in the range of 1 through 15 that specifies a toggle to set.

Considerations

- Write the SETTOG directive as the last item on a directive line.
- If you do not specify any toggle numbers, the SETTOG directive sets all 15 toggles.
- The compiler initially resets all toggles.
- Use the SETTOG and RESETTOG directives with the IF, IFNOT, and ENDIF directives to control conditional compilation.

Example

```
?SETTOG 1,2,5
```

SOURCE Compiler Directive

The SOURCE directive causes the compiler to read source lines from the specified file, either from the beginning of the file to the end of the file, or from the start of a specified section in the file to the end of the section.

```
SOURCE file-name [ ( section [, section ]... ) ]
```

file-name

is the name of a file containing FORTRAN source code. If there are no section names, the file name can be a process, $RECEIVE, a disk file, a terminal, a magnetic tape (unlabeled and unblocked only), or a DEFINE name. Disk files can be structured, unstructured, or EDIT format. If a section name, or a list of section names, is specified, *file-name* must be a disk file. The compiler uses the current default system, volume, and subvolume names for corresponding items omitted from the file name.

section

is the declared name of a section in *file-name*. A section name can be 1 to 31 characters and can be a combination of A through Z, 0 through 9, and the special characters circumflex (^), hyphen (-), and underscore (_). The first character of the name must be a letter.

Considerations

When the compiler has finished reading source lines from the specified file, it resumes reading source lines from the current file.

- The referenced source file can include other SOURCE directives, up to a maximum nesting depth of six levels.

- You must write the SOURCE directive as the last directive if it appears on a line with other directives. You can continue the list of section names on subsequent lines. Each subsequent line must begin with a question mark (?) in column 1.
- At the beginning of each section, the compiler sets the COLUMNS directive value to that specified by the last COLUMNS directive preceding the first SECTION directive in that file.
- When all the specified sections are read, or the end of the file is reached, the compiler resets the COLUMNS directive value to what it was before encountering the SOURCE directive.
- Before reading the first line from the file, and after reverting to the previous file, the compiler prints a line showing the ordinal, file name, and timestamp of the file currently being read. This line is suppressed by the NOLIST and SUPPRESS directives.
- Do not use the SOURCE directive as the first line of an unnamed main program. Name the program using a PROGRAM statement, then write the SOURCE directive.
- If the SOURCE directive does not include a list of section names, FORTRAN reads the entire file, regardless of whether the file contains SECTION directives.

  If the SOURCE directive includes a list of section names, FORTRAN reads only those sections of the file delimited by SECTION directives within the file, and skips all other parts of the file. FORTRAN reads the specified sections in the order in which they physically occur in the file, which is not necessarily the same as the ordering of the section name list in the source directive.

For information about declaring sections in a source program, see the SECTION Compiler Directive on page 10-61.

Examples

The following directive reads the entire file NEWPROG:

```
?SOURCE newprog
```

The following directive reads sections A and C from the file ROUTINES:

```
?SOURCE routines(a,c)
```

SUBTYPE Compiler Directive

The SUBTYPE directive specifies a process subtype for the object file.

The default is SUBTYPE 0.

```
SUBTYPE number
```

number

is an unsigned integer in the range 0 through 63 that specifies a process subtype.

Note. Do not use values 1 through 47 because they are reserved for HP products.

Considerations

- You must specify the SUBTYPE directive on the FORTRAN command line following the semicolon after the object file name, or in the source input file before the first FORTRAN source statement.
- If you specify a SUBTYPE directive after the first FORTRAN statement, the compiler issues an error message and ignores the directive.
- If you specify two or more properly placed SUBTYPE directives, the compiler uses the first one, and issues a warning message for each of the others.
- The SUBTYPE directive specifies the value that a process returns as the device subtype when another process calls the DEVICEINFO system procedure or one of the D-series FILE_GETINFO system procedures (FILE_GETINFO_, FILE_GETINFOBYNAME_, FILE_GETINFOLIST_, and so forth).
- The SUBTYPE directive is effective only for an object file that includes a “main” procedure. You can also use the Binder SET SUBTYPE command to set the process subtype attribute of an object file.

Example

```
?SUBTYPE 52
```

SUPPRESS Compiler Directive

The SUPPRESS directive overrides the effect of the LIST directive; the compiler lists only error messages and compilation statistics.

The default value is NOSUPPRESS.

```
[NO]SUPPRESS
```

Example

```
?SUPPRESS
```

SYMBOLS Compiler Directive

The SYMBOLS directive specifies whether to include a symbol table in the object file for use by Inspect. You must specify this directive if you intend to use Inspect for source-level debugging.

The default value is NOSYMBOLS.

```
[NO]SYMBOLS
```

Considerations

- You can turn the SYMBOLS directive on and off on a procedure by procedure basis. You can delete the symbol table after you have debugged the program.
- Specify the SYMBOLS directive either with the FORTRAN command, or in the source input file preceding the first FORTRAN statement, or between the END line of one program unit and the first FORTRAN statement of the next program unit.
- You can respecify INSPECT, SAVEABEND, and SYMBOLS during an interactive Binder session. For additional information, see the *Binder Manual*.
- You can also specify INSPECT and SAVEABEND as RUN command options.

Example

```
?INSPECT, SAVEABEND, SYMBOLS
```

SYNTAX Compiler Directive

The SYNTAX directive tells the compiler to scan the source file for syntax errors, but does not produce an object file.

```
SYNTAX
```

Considerations

Specify the SYNTAX directive either with the FORTRAN command (after the semicolon following the object file name) or in the source input file before the first FORTRAN statement.

Example

```
?SYNTAX
```

UNIT Compiler Directive

The UNIT directive causes one or more units to exist and declares the properties of the files that will be connected to the units.

```
UNIT { u-lower[- u-upper]                          }
     {(units[, file] [, create-spec]...)           }
```

u-lower
u-upper

is a number in the range 1 through 999. *u-lower* is the lower bound of the range, *u-upper* is the upper bound of the range.

units

is a list of unit ranges in the following form:

```
u-lower [ - u-upper ] [, u-lower [ - u-upper ] ]...
```

file

is an HP file name, DEFINE name, or a network system name.

create-spec

is one of the following:

CODE *file-code*

file-code is an integer in the range 0 through 32767 specifying the Guardian file code to be associated with the file. The default value is 0.

REC *record-size*

record-size is an integer in the range 0 through 32767 that specifies the record size for the file. The default value is 132.

UNITNAME *name*

specifies a name for the unit to be used as the *logical-unit* name in a TACL ASSIGN command. The default value is FT *nnn* where nnn is the unit number as a three-digit decimal integer: FT001 to FT999. name can be from 1 to 31 characters consisting of letters, digits, and hyphens.

access-code

is a keyword that is either INPUT, OUTPUT, or I-O and specifies the allowable access for the unit. The default value is I-O.

exclusion-code

is one of the following keywords: EXCLUSIVE, PROTECTED, or SHARED. *exclusion-code* specifies the file exclusion mode. The default value is

SHARED.

EXT *pri-ext*

EXT (*pri-ext* [, *sec-ext*])

pri-ext and *sec-ext* are integers in the range 0 through 32767 that specify the primary and secondary extent size, in pages. If you omit *sec-ext*, the compiler uses the value of *pri-ext* for *sec-ext*. The default value is (1,1).

Considerations

- You can use one of three methods to connect a unit to a file: the UNIT directive, the TACL ASSIGN command, or the OPEN statement. The order of precedence of these three methods is shown below:

```
?UNIT (7, $s.#1, UNITNAME output) <-- lowest
ASSIGN OUTPUT, $s.#titan
OPEN (7, FILE = '$print') <-- highest
```

  For additional information, see Units on page 5-8.

- You can specify the attributes for a range of units. The following example specifies that the access code for units 8 through 11 is INPUT:

```
?UNIT (8 - 11, INPUT )
```

- For a structured file, you must use the CREATE command of the FUP utility to create the file and specify additional file attributes such as file type and record key descriptions.
- Units 4, 5, and 6 exist automatically, even if there are no UNIT directives defining them.
- Specify the UNIT directive either with the FORTRAN command (after the semicolon following the object file name), or in the source input file before the first FORTRAN statement.

Example

```
?UNIT ( 15-17, spec, REC 80, OUTPUT, PROTECTED )
```

WARN Compiler Directive

The WARN directive instructs the compiler to list compiler warning messages, regardless of the setting of the LIST directive. If you specify NOWARN, the compiler suppresses warning messages.

The default value is WARN.

```
[NO]WARN
```

Example

```
?NOWARN
```

11

Running and Debugging Programs

This section describes how to run a FORTRAN program. It also describes the debugging modes available to you in the Guardian environment. Topics covered in this section include:

Running a FORTRAN Program

The following syntax diagram describes the TACL command to run a FORTRAN program.

```
[RUN[D]] file [/ option [, option ].../ ] [ parameters ]
```

RUN

runs the program whose object code is contained in *file*. The word RUN is optional.

RUND

runs the program whose object code is contained in *file*. RUND specifies that the program is to run in debug mode; it enters the debug state before execution of the first instruction.

file

is the name of a disk file containing the object program to run. The system expands partial file names.

option

is one of the following run-time parameters. For additional run-time options, see the *TACL Reference Manual*.

`IN` *`infile`*

where *`infile`* specifies the name of your input file. If you omit this option, the TACL IN file is used; this is usually the home terminal.

`OUT` *`outfile`*

where *`outfile`* is the name of the output file. If you omit this option, TACL's OUT file is used; this is usually the home terminal.

`NAME [`*`process`*`]`

is the symbolic name assigned to the new process. Name is a "$" followed by up to five alphanumeric characters of which the first must be alphabetic.

Your process must be named if you run it as a process pair. If you specify NAME without *`process`*, the operating system supplies a process name.

If you specify the RUNNAMED compiler directive when you compile your program, your program always runs as a named process. Binder sets the RUNNAMED bit in the object file it creates if any of the object files it includes in the target object file have the RUNNAMED option set. You can also set the RUNNAMED attribute using a Binder command.

If the RUNNAMED attribute is set in file, you do not need to specify the NAME option when you run your program, unless you want the process to have a specific name. For information about the RUNNAMED directive, see RUNNAMED Compiler Directive on page 10-58.

`CPU` *`cpu`*

is an integer ranging from 0 through 15 that specifies the processor in which to run the process.

`PRI` *`pri`*

is an integer ranging from 1 through 199 that specifies the execution priority of the process. Processes with higher numbers execute first. Your system configuration might limit you to a priority that is smaller than 199.

```
INSPECT [ OFF       ]
        [ ON        ]
        [ SAVEBAND  ]
```

sets the debugging environment at run time. INSPECT OFF selects the lowlevel Debug facility. INSPECT ON selects the interactive-symbolic debugger Inspect. INSPECT SAVEABEND is the same as INSPECT ON except that Inspect also automatically creates a save file if the program terminates abnormally.

LIB *library*

where *library* specifies an object file to use as a user library file for this execution of the program. The contents of this file become the library code space for the executing program. Do not specify this option if the object file containing user code space has more than 16 code segments. You can use the LIB option to override the user library file name specified in the FORTRAN LIBRARY directive when you compiled your program.

MEM *pages*

is an integer ranging from 1 through 64 that specifies the maximum number of virtual data pages to allocate for the new process. *pages* overrides the value specified or value estimated by the DATAPAGES compiler directive, and the value estimated by the compiler when you compile the program.

NOWAIT

specifies that TACL return a command prompt after sending the startup message to the new process. If you do not specify NOWAIT, TACL does not return a prompt until the new process completes.

TERM $*name*

is the name of a terminal or process to use as the home terminal for the process. The default terminal is the terminal from which you run your program.

parameters

is one or more parameters that you want to pass to the program in its start-up message. The program must use the GETSTARTUPTEXT routine described in Section 15, Utility Routines, to get these parameters.

The following command runs the object file SROOT using the NUMBERS input file, and sends the listing to a printer:

```
1> RUN sroot/IN numbers, OUT $s.#titan, NOWAIT/
```

The following command runs the object file NEWPROG and selects the Inspect debugger:

```
2> RUN newprog/INSPECT ON/
```

Using TACL PARAM Commands

Table 11-1 shows the TACL PARAM commands that you can specify when you run your program. For more information, see the *CRE Programmer's Guide*.

Table 11-1. Run-Time TACL PARAM Commands

PARAM	Values	Environment	Effect
BUFFERED-SPOOLING	ON OFF	COMMON	BUFFERED-SPOOLING OFF ensures that your program does not use level-3 spooling.
EXECUTION-LOG *name*	* file name	COMMON	If present, affects the names of standard input, standard output, and standard log files.
INSPECT *param*	ON OFF	COMMON	Along with EXECUTION-LOG, determines whether your program invokes a debugger if a run-time error occurs.
NONSTOP *param*	ON OFF	COMMON	Along with other parameters, determines whether your program runs as a NonStop process.
SPOOLOUT	0 1	OLD	Controls whether your program uses level-3 spooling.
SWITCH-*nn*, *value*	*nn* is 1 - 15 value is ON or OFF	COMMON	Specifies the values of software program switches. Programs that specify ENV COMMON can read switch values.

Disabling Level-3 Spooling

Beginning with release C20 of FORTRAN, all program files directed to a spooler collector use level-3 spooling by default. You do not have to change anything in programs written prior to release C20 or recompile them to use this faster method of spooling.

You might want to disable level-3 spooling. By entering a TACL PARAM before you run your program, you can disable level-3 spooling.

Disabling Level-3 Spooling With ENV OLD

If you run a program that you compiled with a C-series FORTRAN compiler or a program that you compiled with a D-series FORTRAN compiler with ENV OLD in effect, you can disable level-3 spooling at run-time by entering a PARAM SPOOLOUT TACL command before you run your FORTRAN program. The general form of the SPOOLOUT PARAM is:

```
PARAM SPOOLOUT {0}
               {1}
```

To disable level-3 spooling, specify:

```
PARAM SPOOLOUT 0
```

Your program uses level-1 spooling for all spooled files, except spooled files for which you explicitly set spooling parameters by calling FORTRANSPOOLSTART. The SPOOLOUT PARAM does not force those spooler files to use level-1 spooling. For more information about FORTRANSPOOLSTART, see Section 15, Utility Routines.

The SPOOLOUT PARAM is meaningful only if you specify ENV OLD.

Disabling Level-3 Spooling With ENV COMMON

The BUFFERED-SPOOLING PARAM is meaningful only if you specify ENV COMMON.

You can enable or disable level-3 spooling by specifying the BUFFERED-SPOOLING PARAM:

```
PARAM BUFFERED-SPOOLING {0}
                        {1}
```

By default, BUFFERED-SPOOLING is ON. To disable level-3 spooling, specify:

```
PARAM BUFFERED-SPOOLING OFF
```

Your program uses level-1 spooling for all spooled files, except spooled files for which you call FORTRAN_SPOOL_OPEN_. The BUFFERED-SPOOLING PARAM does not force those spooler files to use level-1 spooling.

For more information about FORTRAN_SPOOL_OPEN_, see the FORTRAN_SPOOL_OPEN_ Routine on page 15-11.

Using the EXECUTION-LOG PARAM

The EXECUTION-LOG PARAM is meaningful only if you specify ENV COMMON.

The EXECUTION-LOG PARAM affects the names of the files that FORTRAN uses for standard input, standard output, and standard log if you are using the shared file facilities in D-series FORTRAN. For more information about file sharing in D-series FORTRAN, see Using Shared Files on page 13-27.

The EXECUTION-LOG PARAM and Standard Input

FORTRAN determines the file name for standard input as follows. If the INFILE name in your program's startup message is:

- Not the name of your program's home terminal, FORTRAN uses the INFILE name from the startup message for standard input.
- Blanks, FORTRAN does not open a system file but accepts open requests and returns end of file each time your program reads from standard input.
- The name of your program's home terminal and
  - You do not specify the EXECUTION-LOG PARAM, FORTRAN opens your home terminal if your program opens standard input.
  - You specify a file name as the EXECUTION-LOG, FORTRAN uses the file you specify for the EXECUTION-LOG as standard input.

    For example, if your home terminal is named $TERM and you specify AFILE for EXECUTION-LOG, FORTRAN opens AFILE if a routine in your program opens standard input:

    ```
    PARAM EXECUTION-LOG AFILE
    RUN myprog / IN $TERM, ..... /
    ```

    If you do not specify the IN parameter, TACL passes the name of your terminal as the IN parameter, which has the same effect as explicitly specifying your home terminal as the IN parameter:

    ```
    PARAM EXECUTION-LOG AFILE
    RUN myprog
    ```

  - You specify an asterisk for the EXECUTION-LOG, FORTRAN does not open a file for standard input. Instead, FORTRAN returns end of file each time your program reads from standard input:

    ```
    PARAM EXECUTION-LOG *
    RUN myprog / IN $TERM, ..... /
    ```

    If you do not specify the IN parameter, TACL passes the name of your terminal as the IN parameter, which has the same effect as explicitly specifying your home terminal as the IN parameter:

    ```
    PARAM EXECUTION-LOG *
    RUN myprog
    ```

The EXECUTION-LOG PARAM and Standard Output

FORTRAN determines the file name for standard output as follows. If the OUTFILE name in your program's startup message is:

- Not the name of your program's home terminal, FORTRAN uses the OUTFILE name from the startup message for standard output.
- Blanks, FORTRAN does not open a system file but accepts open requests, discards records that you write to standard output, and indicates a successful write each time your program writes to standard output.
- The name of your program's home terminal and You do not specify the EXECUTION-LOG PARAM, FORTRAN opens your home terminal if your program opens standard output.
  - You specify a file name as the EXECUTION-LOG, FORTRAN uses the file you specify for the EXECUTION-LOG as standard output.

    For example, if your home terminal is named $TERM and you specify AFILE for EXECUTION-LOG, FORTRAN opens AFILE if a routine in your program opens standard output:

    ```
    PARAM EXECUTION-LOG AFILE
    RUN myprog / OUT $TERM, ..... /
    ```

  - You specify an asterisk for the EXECUTION-LOG, FORTRAN does not open a file for standard output. Instead, each time your program writes to standard output, FORTRAN discards the record and indicates a successful write:

    ```
    PARAM EXECUTION-LOG *
    RUN myprog / OUT $TERM, ..... /
    ```

The EXECUTION-LOG PARAM and Standard Log

The EXECUTION-LOG PARAM specifies a file name to which FORTRAN might write diagnostic messages. The syntax of the EXECUTION-LOG PARAM is:

```
PARAM EXECUTION-LOG { filename }
                    { *        }
```

By default, FORTRAN writes log messages to your process's home terminal. You can direct log messages to another file, however, by specifying a file name in a TACL ASSIGN command or in a PARAM EXECUTION-LOG command.

- If an ASSIGN specifies the logical name STDERR, FORTRAN uses the physical name from the ASSIGN as the name of standard log.
- If a PARAM specifies EXECUTION-LOG, FORTRAN uses the value of the EXECUTION-LOG PARAM as the name of standard log. If the EXECUTION-LOG

PARAM specifies an asterisk (*), FORTRAN does not write messages to standard log.

- If an ASSIGN specifies STDERR and a PARAM specifies EXECUTION-LOG, FORTRAN uses the physical name from the ASSIGN unless the physical name specifies your home terminal, in which case FORTRAN uses the value of the EXECUTION-LOG PARAM or, if the EXECUTION-LOG PARAM value is an asterisk, FORTRAN does not open standard log.

FORTRAN does not open a Guardian file for standard log if the ASSIGN for STDERR specifies the process's home terminal, and the value of the EXECUTION-LOG PARAM is an asterisk.

Using Debug Facilities

HP supports two debugging programs:

- Debug—a low-level debugger
- Inspect—an interactive, symbolic debugger

Your program uses the Debug program:

- If you specify INSPECT OFF in the RUN or RUND command, as follows:

  ```
  RUND program / INSPECT OFF/
  ```

- By default if you have selected Inspect but the necessary support processes are not running on the system on which the process to be debugged is running.

You select the Inspect facility in one of the following ways:

- By specifying the INSPECT or SAVEABEND compiler directives

  ```
  ?INSPECT
  ```

  ```
  ?SAVEABEND
  ```

- By using the Binder SET INSPECT or SET SAVEABEND commands during a Binder session

  ```
  SET INSPECT ON
  ```

  ```
  SET SAVEABEND ON
  ```

- By using the TACL SET INSPECT command prior to the RUN command that starts the process

  ```
  SET INSPECT ON
  ```

- By selecting the INSPECT ON or INSPECT SAVEABEND options of the RUN command when you run your program

  ```
  RUND program / INSPECT ON/
  ```

If you use more than one method of selecting a debugger, precedence is established as follows: A program file value (specified either in a compiler directive—INSPECT or NOINSPECT—or a Binder command—SET INSPECT ON or SET INSPECT OFF) overrides the value specified in the RUN command, which overrides the value specified in a TACL session.

Using the INSPECT TACL PARAM

If an error occurs in your FORTRAN program—for example, you attempt to divide by zero—the FORTRAN run-time routines write an error message to the standard log file and, if you specify the INSPECT PARAM before you run your program, invoke a debugging program.

The INSPECT PARAM is meaningful only if you specify ENV COMMON.

The following is the syntax of the INSPECT PARAM:

```
PARAM INSPECT { ON  }
              { OFF }
```

The default is INSPECT OFF.

If you specify PARAM INSPECT ON, and certain run-time errors occur, FORTRAN invokes a debugger program—Inspect or Debug—rather than terminating your program immediately.

If you specify PARAM INSPECT OFF or you do not specify an INSPECT PARAM, your program terminates immediately.

The INSPECT PARAM affects only those programs that you run after you set the PARAM. It does not affect programs that you have already started.

For more information about choosing the Debug program or the Inspect program for debugging, see Using Debug Facilities on page 11-8.

Using Inspect

Inspect is an interactive-symbolic debugger that lets you control program execution, display values, and modify values in terms of source-language symbols. Inspect has two basic modes: high-level and low-level.

Note. Unlike FORTRAN, Inspect does not allow embedded spaces in identifier names. You must enter each identifier name as a consecutive string of nonblank characters.

High-Level Inspect

Using high-level Inspect you can:

- Identify code and data locations using source language expressions.
- Assign values to data locations, and display values from data locations in a specific format.
- Step through program execution by language-oriented or machine-oriented increments.
- Display source program text surrounding the currently executing statement.
- Define names for Inspect command strings.
- Direct Inspect to suspend program execution and perform a specified action whenever a certain code location is reached, or whenever a certain data item is manipulated.
- Save a copy of your process environment image in a disk save file.

You must specify the FORTRAN SYMBOLS compiler directive when you compile your program if you want to use high-level Inspect.

Low-Level Inspect

Using Inspect in low-level mode is very similar to using the Debug program. Lowlevel Inspect offers the following additional features compared to the Debug program:

- Recognition of source-language procedure names
- Display in ICODE for machine instructions

Low-level Inspect enables you to display and modify registers. You do not need symbol information to use low-level Inspect.

When Inspect encounters a program unit that was compiled without the SYMBOLS directive, it automatically enters low-level mode. You can also place Inspect in lowlevel mode by using the Inspect LOW command.

When you use low-level Inspect, you can use features of high-level Inspect that do not require symbols. These enable you to:

- Inquire about or set the program environment (the default system, volume, and subvolume)
- Use the FC command to edit a previous command
- Display help information
- Use the OBEY command to read commands from a disk file
- Display machine instructions expressed as ICODE
- Step through program execution by one or more machine instructions
- Exit from Inspect

For additional information about Inspect commands in general, and Inspect support for FORTRAN in particular, see the *Inspect Manual*.

Using the NONSTOP PARAM

The NONSTOP PARAM is meaningful only if you specify ENV COMMON.

You can use the NONSTOP PARAM to specify whether you want your program to run as a NonStop process. The syntax of the NONSTOP PARAM is

```
PARAM NONSTOP { ON  }
              { OFF }
```

The default is PARAM NONSTOP ON.

If you specify PARAM NONSTOP OFF, your program does not run as a NonStop process. If you specify PARAM NONSTOP ON, your program runs as a NonStop process only if your program also specifies the NONSTOP directive and you execute a START BACKUP statement.

Using SWITCH-nn PARAM

The SWITCH- nn PARAM is meaningful only if you specify ENV COMMON.

You use the SWITCH-*nn* PARAM to turn on or off logical program switches that your program can read at run time. The syntax of the SWITCH- *nn* PARAM is

```
PARAM SWITCH- nn { ON  }
                 { OFF }
```

The default for all switches is OFF.

You can set switch values for 15 switches, identified as SWITCH-1 through SWITCH-15. Only switch values that you specify before you run your program are meaningful.

Changing the values of switches after your program begins running does not affect the switch values used by your program.

You read switch values using the SSWTCH utility routine, described in Section 15, Utility Routines.

12 Memory Organization

Topics covered in this section include:

The information in this section might be useful to you:

- If you need to control where FORTRAN allocates data and data blocks
- If you are using Inspect or Debug to debug a program
- If you are combining FORTRAN programs written in languages other than FORTRAN

Code Space

The code area of a process consists of a user code space and an optional user library space. Each space can have up to 16 code segments of 64K words, or a total of up to 1024K words in each of the two code spaces.

If an object program is executed with a user library, each code space is an object file containing up to 16 code segments. You run the object file that includes the main program (this file becomes the user code space), using the LIBRARY directive or the LIB run-time option to specify the object file that becomes the user library code space.

If a program is executed without a user library, the object file can have up to 32 code segments, for a total code size of up to 2048K words. The operating system allocates the first 16 code segments to the user code space and any remaining code segments to the user library space.

A program is said to use extended code space if it uses more than a one segment.

You don’t need to do anything special to write and compile programs that use extended code space. FORTRAN compiles the program units and calls Binder to combine them in a single object program. Binder produces an object program that uses as many code segments as needed, up to the limit of 32 segments.

Information in a code segment consists of instruction codes and program constants.

Your program can read the contents of a code segment but the TNS hardware reports an instruction trap if your program attempts to write to a code segment. Therefore, you cannot modify code segments during execution.

A code segment consists of up to 65,536 16-bit words which are numbered consecutively from C[0] (code, element 0) through C[65535].

The P (program) register is the program counter. It contains the 16-bit C[0]-relative address of the current instruction plus one. The contents of the P register are incremented by one at the beginning of each instruction that your program executes so that, normally, instructions are fetched (and executed) from ascending memory locations.

Data Space

FORTRAN allocates memory for your program's data

- In the lower half of the user data segment
- Depending on the directives you specify, in the upper half of the user data segment
- In your program's extended memory segment

FORTRAN also allocates space for file buffers, file control blocks and other internal data structures in the user data segment. The user data segment consists of up to 65,536 16-bit words. Addresses in the data segment start at G[0] (global data, word 0) and progress consecutively through G[65535].

Figure 12-1 on page 12-3 shows how FORTRAN allocates data for your program if you specify ENV OLD. Figure 12-2 on page 12-4 shows how FORTRAN allocates data for your program if you specify ENV COMMON.

The lower half of the user data segment contains global data and the run-time stack on which FORTRAN dynamically allocates storage for local data when your program calls subprograms. This area is called the memory stack and is logically separated into three areas: global, local, and sublocal ("top of the stack").

You can address data within the global area by an instruction in a program. The addressing base is G[0]. FORTRAN stores statically allocated data in the global area, including:

- I/O control blocks, unless the HIGHCONTROL directive is present.
- OWN data blocks, containing local variables, arrays, and RECORDs that are named in DATA or SAVE statements.
- Common data blocks, unless HIGHCOMMON or LARGECOMMON directives specify otherwise.

Data within the local area is known only to the currently executing procedure and its subprocedures. The beginning of the local area is defined by the value in the 16-bit L register when your program begins execution. The L (local) register contains the G[0]-relative address of the word at the beginning of this area. The addressing base for local data is L[0]. FORTRAN stores dynamically allocated local data in this area, including local variables, arrays, and RECORDs that are not named in DATA or SAVE statements.

Data above the current L register is known only to the currently executing procedure. The 16-bit S register contains the G[0]-relative address of the last word currently

defined in the memory stack. All local data in your program is referenced relative to the L register except the dummy arguments to statement functions, which are referenced relative to the S register.

Word-addressable and byte-addressable data can be stored in the memory stack area.

Figure 12-1. User Data Segment for ENV OLD

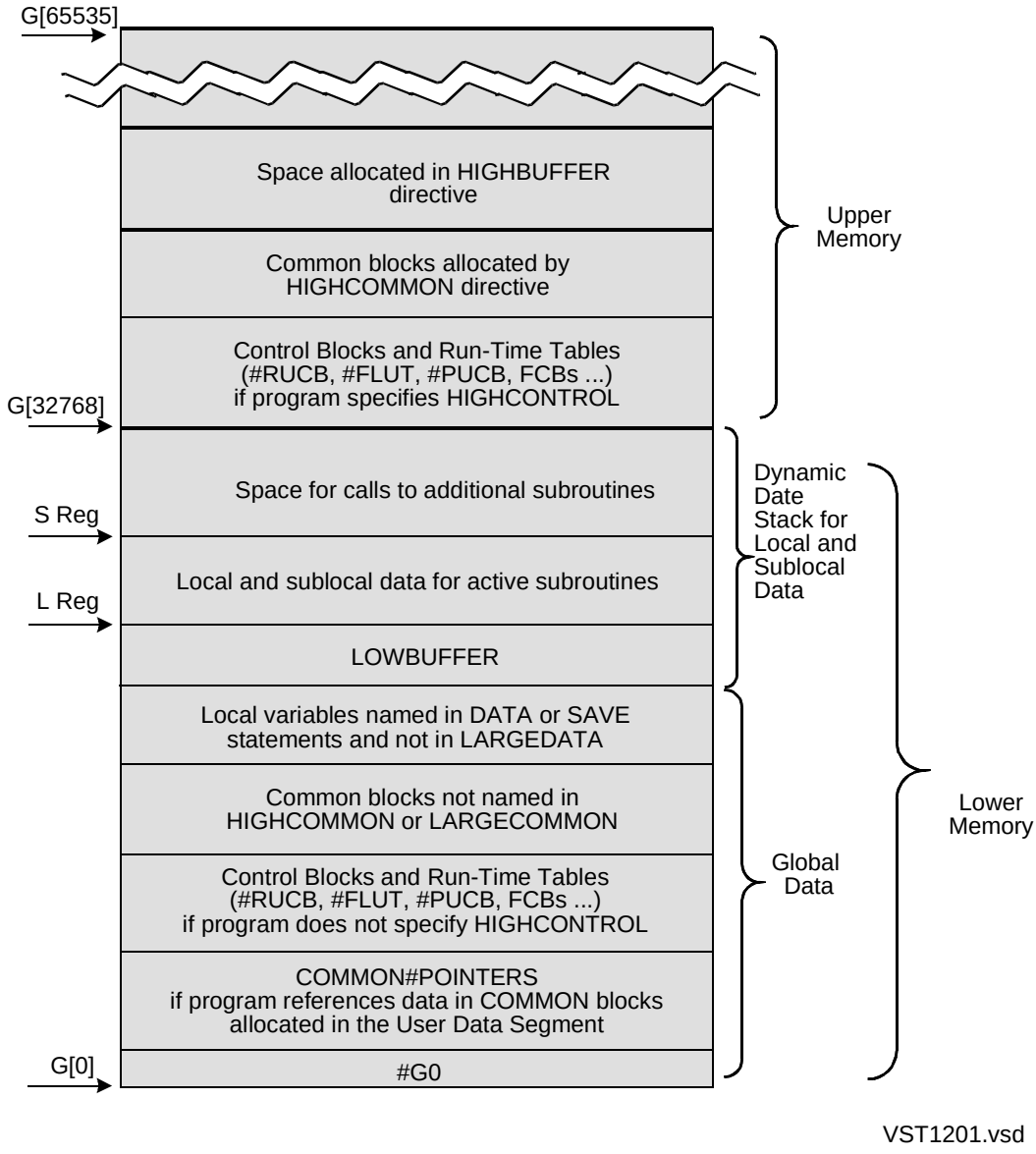
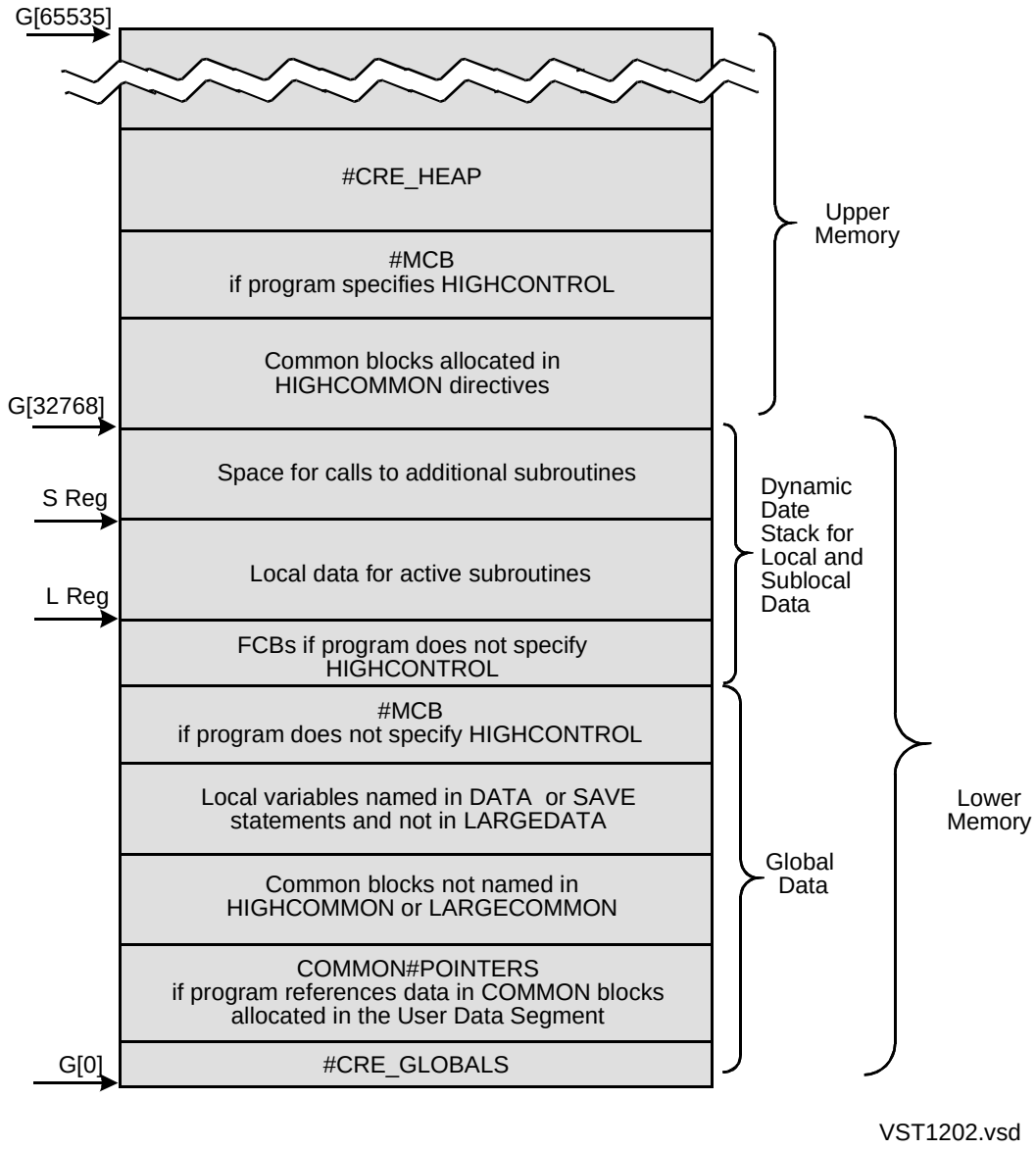


Figure 12-2. User Data Segment for ENV COMMON



Three logical areas within the user data segment are available to a FORTRAN subprogram while it is active:

- COMMON, DATA, and SAVE areas, using G-plus addressing
- Subprogram parameter (argument) area, using L-minus addressing
- Local area, using L-plus addressing

Upper Memory

The upper memory area (the “upper” 32K words of the user data segment) is optionally available for

- Common data specified by the HIGHCOMMON directive
- Control information specified by the HIGHCONTROL and HIGHBUFFER directives

Access to upper memory is by indirect addressing only. FORTRAN stores only word-addressable data in upper memory.

Storage Areas

The Binder load map in your program listing includes a list of the data blocks that FORTRAN allocates to hold program data as well as data blocks that hold internal data used by the FORTRAN run-time environment. The names and contents of these data blocks depend on whether you compile your program with ENV COMMON or ENV OLD in effect. Table 12-1 lists the data blocks allocated by FORTRAN. Table 12-2 on page 12-6 lists the compiler directives that affect how FORTRAN allocates data.

Table 12-1. Data Blocks

Block	OLD	COMMON	Directives That Affect Block’s Location and Size
#FLUT	X	–	HIGHCONTROL
#PUCB	X	–	HIGHCONTROL
#RUCB	X	–	HIGHCONTROL
FCBs	–	X	HIGHCONTROL
#HIGHBUF	X	–	HIGHBUFFER
#LOWBUF	X	–	LOWBUFFER
#G0	X	–	none
#MCB	–	X	HIGHCONTROL, HIGHBUFFER
#CRE_HEAP	–	X	HIGHBUFFER
#CRE_GLOBALS	–	X	none
Common blocks	X	X	HIGHCOMMON, LARGECOMMON
COMMON#POINTERS	X	X	LARGECOMMON
#RECEIVE	–	X	RECEIVE

X The data block is used in the specified environment

– The data block is not used in the specified environment

Table 12-2. Compiler Directives That Control Data Allocation (page 1 of 2)

	Effect	
Directive	**ENV OLD**	**ENV COMMON**
DATAPAGES	Specifies how many pages to allocate for data storage in the user data segment (Default: Compiler estimates)	Always uses 64
EXTENDCOMMON	Allocate one pointer for each common block. (Default: allocate one pointer for each variable referenced in a common block)	Same as ENV OLD
EXTENDEDREF	Generates code that uses doubleword addresses for parameters to subprograms. (Default: use word addresses)	Same as ENV OLD
HIGHBUFFER *n*	Allocate *n*-byte #HIGHBUF in upper memory (Default: no #HIGHBUF)	Allocate n-byte #CRE_HEAP in upper memory (Default: allocate 1,024-byte #CRE_HEAP in upper memory)
HIGHCOMMON *blks*	Allocate all or specified common blocks in high memory. If blks not specified, allocate all common blocks in high memory. See also LARGECOMMON directive in Section 10, Compiler Directives. (Default: allocate common blocks in low memory or extended memory according to LARGECOMMON directives)	Same as ENV OLD
HIGHCONTROL	Allocate #RUCB, #FLUB, #PUCB and FCBs in high memory (Default: allocate blocks in low memory)	Allocate #MCB and FCBs in high memory (Default: allocate #MCB and FCBs in low memory)
LARGECOMMON *blks*	Allocate all or specified common blocks in extended memory (Default: Allocate common blocks in low memory or high memory according to HIGHCOMMON directives)	Same as ENV OLD
LARGEDATA *items*	If *items* is a constant, allocate all local data with length greater than or equal to *items*, in extended memory. If *items* is not a constant, allocate items in extended memory	Same as ENV OLD

Table 12-2. Compiler Directives That Control Data Allocation (page 2 of 2)

	Effect	
Directive	**ENV OLD**	**ENV COMMON**
LARGESTACK *n*	Allocate *n* bytes of data in extended memory for data specified in LARGEDATA directives	Same as ENV OLD
LOWBUFFER *n*	Allocate *n*-byte #LOWBUF (Default: #LOWBUF is 512 bytes)	not used
RECEIVE *params*	Allocate space for $RECEIVE information in #PUCB	Allocate #RECEIVE in low memory (Default: do not allocate #RECEIVE)

CONTROL Storage Areas With ENV OLD

If you compile your program with ENV OLD in effect, the Binder load map printed for a FORTRAN program compilation shows three “special data blocks” named #RUCB, #FLUT, and #PUCB. They are in the upper half of the user data segment if the HIGHCONTROL directive is used, or in the lower half otherwise. These blocks contain information used by the FORTRAN run-time library procedures that do the work of FORTRAN I/O statements and handle errors detected by intrinsic functions. They are also used by the Saved Message Utility procedures.

- The Run-Unit Control Block (#RUCB)

  The #RUCB block contains pointers to the other control blocks and to the buffer storage areas. It always contains 77 words.

- The FORTRAN Logical Unit Table (#FLUT)

  The #FLUT block contains pointers to the File Control Blocks (FCBs) for all the FORTRAN program’s I/O units. Its size is N + 2 words, where N is the largest I/O unit number that exists in the compilation.

- The Program Unit Control Block (#PUCB)

  The #PUCB block contains a 39-word fixed header area, plus a File Control Block (FCB) area for each I/O unit that exists for the compilation, with 38 to 58 words per FCB depending on the lengths of the file names and unit names.

BUFFER Storage Areas With ENV OLD

The Binder load map includes another “special data block” named #LOWBUF, and might include another data block named #HIGHBUF. Both of these are buffer pools used as “heap storage” by several of the FORTRAN run-time library procedures for data areas that must be allocated and released dynamically as program execution

proceeds, but that cannot be on top of the run-time stack because they must remain allocated after the procedure creating them has returned to its caller.

The #LOWBUF area contains space for the Saved Message Utility storage area, plus the $RECEIVE file message queue area, plus a general area for file buffers. Given the directives:

```
? LOWBUFFER b
? SAVE (STARTUP s, ASSIGNS a, PARAM p)
? RECEIVE (OPEN m, MAXREPLY r, SYNCDEPTH d, QDEPTH q)
```

the size of the #LOWBUFFER area is:

```
b + 1 +
s/2 + 3 + a * 57 + p/2 + 3 +
(m + 1) * 11 + ((r + 1)/2 + 4) * m * d + q * 4
```

words, where the default values are:

`b = 512` if not specified in a LOWBUFFER directive

`s = 0, a = 0, p = 0` if not specified in a SAVE directive

`m = 1, r = 132, d = 1, q = 1` if not specified in a RECEIVE directive

Thus, in the absence of these directives, the #LOWBUFFER area is allocated 609 words.

The #HIGHBUFFER area contains the number of words specified in the HIGHBUFFER directive, or is omitted if none is specified. During program execution, the FORTRAN run-time library procedures call the GPLIB procedures GET^BUFFER and PUT^BUFFER to allocate and release space in the buffer pool consisting of the #LOWBUFFER and #HIGHBUFFER areas of memory. The FORTRAN run-time system allocates and de-allocates space in this buffer pool as follows:

- The Saved Message Utility area is allocated when the program begins execution. It must be in the #LOWBUFFER area. Its size depends on the startup, assign, and param system messages the process receives, and may be less than or greater than the amount specified in the SAVE directive. This area might become larger or smaller as the program calls Saved Message Utility routines to add, delete, and alter saved messages.
- The $RECEIVE area is allocated when the file named $RECEIVE is opened. It can be in either the #LOWBUFFER or #HIGHBUFFER area. Its size is exactly as specified in the RECEIVE directive, or 96 words by default.
- When a FORTRAN program is running as a NonStop process, the system allocates 169 words in the #LOWBUFFER area for each EDIT format file that is open.
- The system may allocate 512 words for each output file that is open for level-3 spooling. This can be in the #LOWBUFFER or #HIGHBUFFER area. It is not

allocated by the system if sufficient space is not available or if the program specifies a level-3 buffer area with a valid address in a call to FORTRANSPOOLSTART. The program can run correctly, though slower, without a level-3 spooling buffer area.

User-written TAL subprograms can also call GET^BUFFER and PUT^BUFFER to make further use of these areas. This means that, with the default #LOWBUFFER size of 609 words:

- A FORTRAN program running as a NonStop process can have up to three EDIT format files open at a time, because 3 * 169 = 507.
- A FORTRAN program that is not running as a NonStop process or has no EDIT format files open can have one output file open with automatic level-3 spooling, because each level-3 buffer is 512 words.

Storage Areas With ENV COMMON

If ENV COMMON is in effect, FORTRAN allocates three special data blocks named #CRE_GLOBALS, #CRE_HEAP, and #MCB. #CRE_GLOBALS and #MCB contain run-time information for programs that specify the common environment. The runtime environment uses space from #CRE_HEAP for file buffers and so forth. The default size of #CRE_HEAP is 1,024 bytes, but you can specify its size with the HIGHBUFFER compiler directive. You should not specify a value that is less than 1,024 for HIGHBUFFER.

FORTRAN allocates a data block for each unit you define in your program. Each data block holds one file control block. The names of these data blocks are of the form ##FT*nnn* where *nnn* is the unit number.

If you specify a RECEIVE directive, FORTRAN allocates a data block called #RECEIVE.

If you specify HIGHCONTROL, FORTRAN allocates the #MCB data block and all file control blocks in upper memory.

#CRE_HEAP is always in upper memory.

#CRE_GLOBALS and #RECEIVE are always in low memory.

FORTRAN does not allocate #LOWBUF in the common environment.

Storage of Entities in Common Blocks

When a program unit transfers control to a subprogram, it can also transfer data through the subprogram’s arguments. Similarly, the called subprogram can return data to its caller through its arguments. Function subprograms also return data as the value of the function. These are the only means of exchanging data between subprograms without using common blocks.

When a subprogram returns control to the calling program unit, its local data is lost because local data is allocated on the stack. The COMMON statement enables you to

place entities in global storage; any program unit that makes reference to common storage can use these entities. You can also use DATA and SAVE statements to place specified entities in global storage and to store their addresses in the code area.

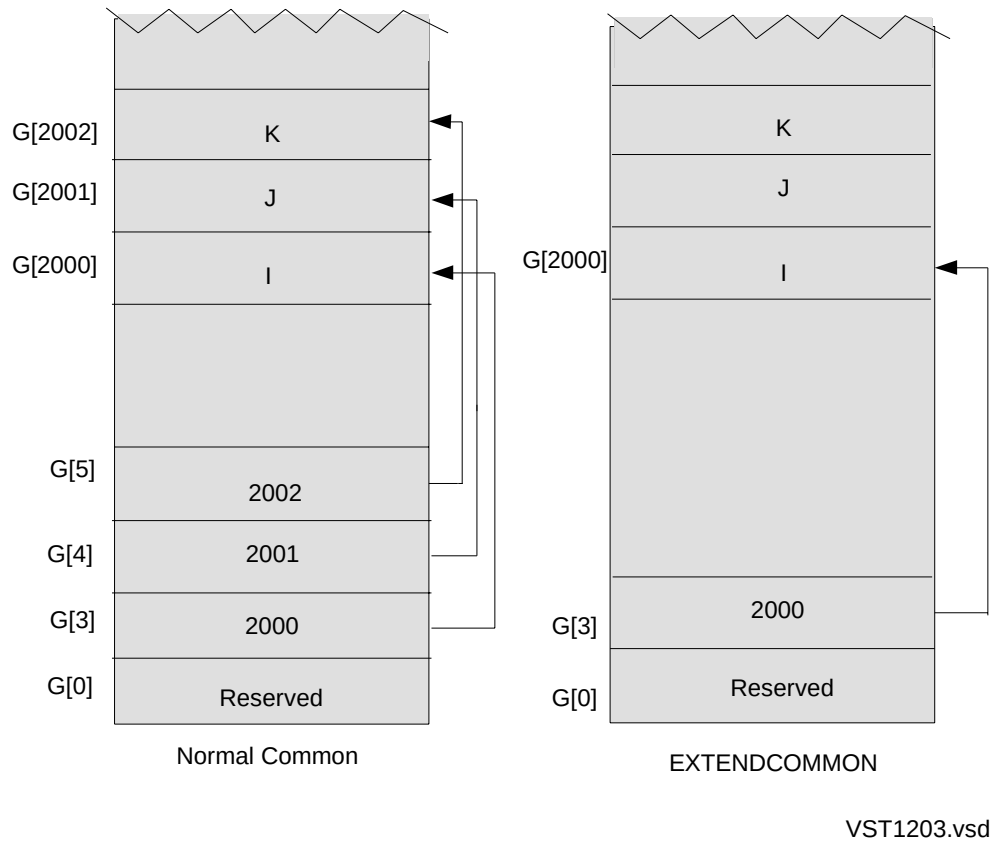
Three compiler directives affect the addressing and storage of entities in common:

- EXTENDCOMMON, which provides indexed indirect addressing. (Note that this directive affects only word and doubleword variables; it has no effect on arrays.)
- LARGECOMMON, which allocates space for common blocks in extended memory. (For more information, see the Extended Memory on page 12-11.)
- HIGHCOMMON, which allocates space for common blocks in upper memory.

Figure 12-3 shows the difference between using the default common addressing and EXTENDCOMMON addressing for entities stored in common blocks by the following statement:

```
COMMON i, j, k
```

Figure 12-3. Normal and EXTENDCOMMON Addressing



By default, FORTRAN allocates a pointer in primary global storage for each entity in a common block in secondary global storage. If you specify the EXTENDCOMMON directive, FORTRAN allocates only one pointer for each array and one pointer for all scalars in the common block. The latter pointer points to the first entity in the block; the

entities that follow are located by indexing. Having one pointer for all scalars saves space in primary global storage, at some expense for additional processing time.

Extended Memory

An object program executing on an HP NonStop system always has exactly one user data segment of up to 65,536 words (addressable with 16-bit word addressing) and can also have any number of extended data segments of up to 127.5MB (addressable with 32-bit byte addressing). A FORTRAN program can directly access just one extended data segment.

The LARGECOMMON directive specifies which common blocks to allocate in the extended data segment. Common blocks are always statically allocated. Common blocks that are in the extended data segment have no pointers in the global area of the user data segment.

The LARGEDATA directive specifies which local data items to allocate in the extended data segment. “Data items” means variables, arrays, and RECORDs. “Local data items” means data items that are local to a program unit (they are not dummy arguments, not in a common block, and not equivalenced to anything in a common block).

Local data items are statically allocated (they have a fixed memory location throughout program execution) if they are named in DATA or SAVE statements.

Otherwise, they are dynamically allocated on a run-time stack when their procedure is entered and de-allocated when that procedure returns to its caller.

Statically Allocated Data

Statically allocated local data items are contained in OWN data blocks. Each program unit can have zero, one, or two OWN data blocks. If a statically allocated local data item is in extended memory because of a LARGEDATA directive, it is in an OWN data block in the extended data segment; the block name is an ampersand (&) followed by the program unit name. Other statically allocated local data items are in an OWN block in the global area of the user data segment; the block name is a plus sign (+) followed by the program unit name.

Dynamically Allocated Data

Dynamically allocated local data items are in one of two run-time stack areas. If a dynamically allocated local data item is in extended memory because of a LARGEDATA directive, it is in the extended stack data block whose name is $EXTENDED#STACK. Other dynamically allocated local data items are in the local area of the run-time stack in the user data segment.

The hardware L and S registers point to the beginning and end of the currently executing procedure’s local area in the extended data segment run-time stack. Pointer doublewords in the global area of the user data segment serve a similar purpose.

These pointers are in the data blocks named EXTENDED#STACK#FRAME and EXTENDED#STACK#POINTERS.

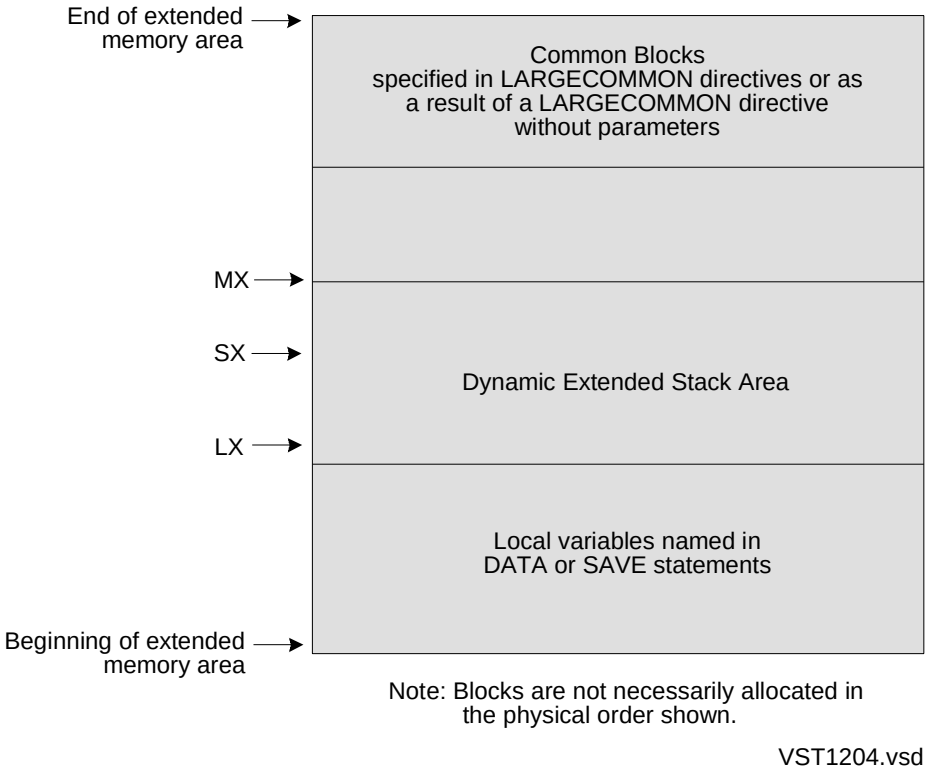
The user data segment run-time stack cannot grow past address G[32767]. If a program exceeds this limit, it is interrupted with a stack overflow trap, and the system calls Debug or Inspect unless the ARMTRAP procedure has been called.

Whether you specify ENV OLD or ENV COMMON, your program always traps if it overflows the run-time stack. Your program retains control only if you have defined your own trap handler. If your program specifies ENV OLD, TACL writes a trap 3 error message to your terminal. If your program specifies ENV COMMON, the FORTRAN run-time library writes a message to your terminal specifying trap error 5.

Similarly, the extended data segment run-time stack cannot grow past the size specified for it when the object program file is created. The compiler estimates the proper size for this area. You can use the LARGESTACK compiler directive or the Binder SET LARGESTACK command to override the value computed by the compiler. If the program attempts to exceed this size limit, the FORTRAN run-time system displays a message on the home terminal and calls the Guardian procedure ABEND if your program specifies ENV OLD or PROCESS_STOP_ with the ABORT option set if your program specifies ENV COMMON.

Figure 12-4 shows the allocation of space in the extended data segment. The pointers shown here as LX, SX, and MX are contained in the two special data blocks mentioned above.

Figure 12-4. Extended Data Segment



Debugging Programs That Use Extended Memory

You can use Inspect or Debug to debug programs that use extended memory.

If you use high-level Inspect commands to debug your programs, you won't notice any changes when you debug programs with extended memory. Because Inspect allows you to refer to program entities by symbolic names (rather than memory addresses), using extended memory does not affect the way you use Inspect.

If you use Debug or low-level Inspect commands to debug programs that use extended memory, you must specify segment numbers as part of memory addresses.

See the *Debug Manual* to find out how to specify segment numbers in Debug commands. See the *Binder Manual* to find out how to use Binder listings to determine the segment numbers of entities in your program.

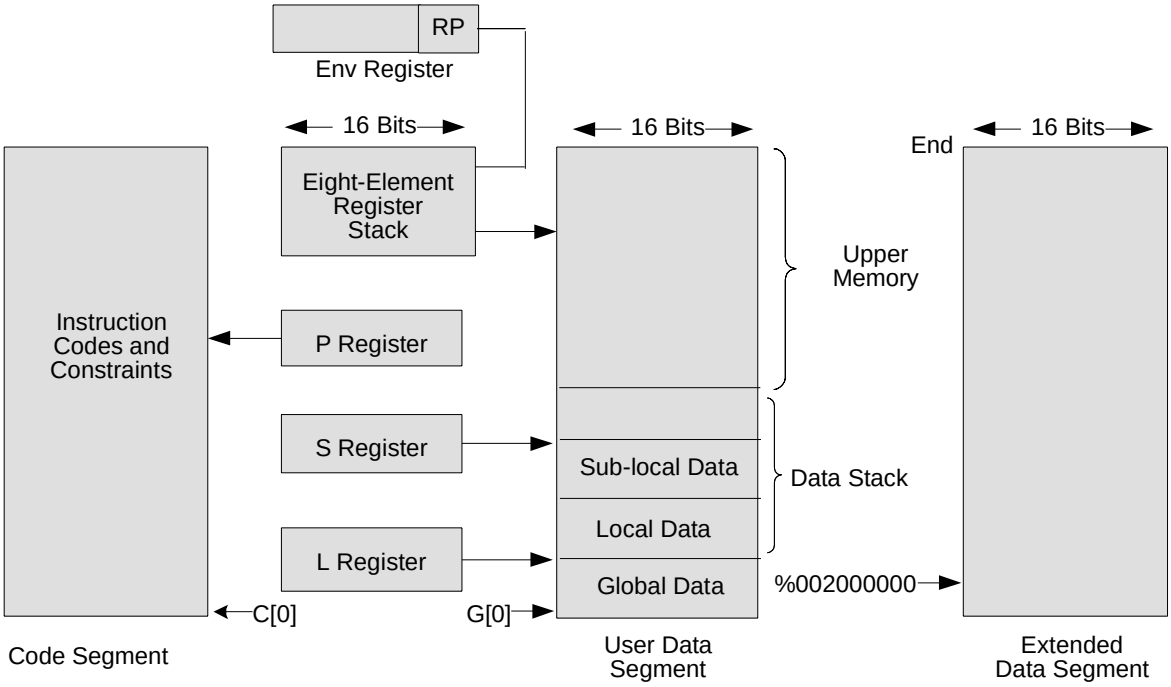
TNS Processor Memory Organization

Program memory on a TNS processor consists of:

- A code area, containing the program instruction codes and constants
- A data area—the user data segment
- An optional extended data area
- Four registers (P, L, S, and E)
- An eight-element register stack used for computation (three elements are available for address indexing)

For more information about processor organization, see the system description manual for your processors.

Figure 12-5. Program Memory Environment



P Register:	Program counter: address of current instruction plus one (relative to C[0])
Global Data:	Data area accessible from any point in a program
Local Data:	Data area accessible only from currently executing procedure and its subprocedures
Sub-Local Data:	Data area accessible only from currently executing statement function
L Register:	Local data pointer: G[0]-relative address of first element in the local data area; also indicates the location in the memory stack of the link (stack marker) back to the calling procedure
S Register:	Top of stack: G[0]-relative address of the last active element in the memory stack
Register Stack:	Eight-element register stack where arithmetic operations are performed.Three elements can also be used for indexing.
Register Stack:	Points to—holds the address of—the current top of the register stack.
Pointer—RP:	The stack is empty if RP = 7

VST1205.vsd

Accessing Data

Data is accessed through memory reference instructions. Locations in the user data segment can be addressed through:

- The address field in a memory reference instruction (direct addressing)
- An address pointer in memory (indirect addressing)
- An offset value to be added to a direct or indirect address (indexed addressing)

13 Mixed-Language Programming

This section describes how you can combine procedures written in C, COBOL85, FORTRAN, Pascal, and TAL into an executable object file.

Topics covered in this section include:

You might find mixed-language programming useful because each language offers distinct advantages in writing certain kinds of routines or because you don't want to rewrite code written in another language that you want to use in a FORTRAN program.

For information about data type correspondence in C, COBOL85, FORTRAN, Pascal, and TAL, see Appendix D, Data Type Correspondence and Return Value Sizes.

Prior to D-series software, object files written in one HP language could be bound with object files from other HP languages but they were extremely limited in the resources they could use. With few exceptions, all run-time libraries used the same memory areas but each run-time library specified its own unique layout for the data in that area.

The Common Run-Time Environment—CRE

D-series software introduces the facilities of the Common Run-Time Environment (CRE). The CRE provides services to programs written in C, COBOL85, FORTRAN, Pascal, and TAL.

Using the CRE

In general, you do not need to do anything special to take advantage of the services of the CRE. The FORTRAN run-time library accesses routines in the CRE when it is appropriate to do so—you need to change few, if any, constructs in your program to take advantage of the services of the CRE.

If you specify ENV COMMON when you compile your FORTRAN program, routines in the FORTRAN run-time library call CRE routines for some of the services that are handled by Guardian routines in C-series software. Programs that specify ENV OLD do not call CRE routines.

The FORTRAN run-time library uses CRE services to:

- Manage shared access to unit 5 and unit 6
- Manage NonStop process pairs
- Write run-time diagnostic messages
- Manage hardware traps
- Manage $RECEIVE
- Support intrinsic functions

For more information about the CRE, see the *CRE Programmer's Guide*.

Sharing Files When ENV COMMON Is in Effect

This subsection describes how your FORTRAN program shares access to the files connected to unit 5 and unit 6 if you specify ENV COMMON.

If you specify ENV COMMON, your FORTRAN routines can share access to the files connected to units 5 and 6 with other routines in your process, even if the other routines are written in languages other than FORTRAN. The files associated with units 5 and 6 are referred to generically as standard input and standard output, respectively. (The common environment of D-series software also supports shared access to a log file—standard log—but you cannot establish a unit connection to the standard log file from a FORTRAN routine. FORTRAN writes the message that you specify in PAUSE and STOP statements and in the FORTRAN_COMPLETION_ utility to the standard log file and also writes diagnostic messages to the standard log file.)

The D-series FORTRAN run-time routines treat all other unit connections just as C-series run-time routines do. Routines written in C, COBOL85, FORTRAN, Pascal, and TAL can share a single file open, but only for the standard files. In FORTRAN, the standard files are unit 5 (standard input) and unit 6 (standard output). In COBOL85, you access the standard files by executing ACCEPT (standard input) and DISPLAY (standard output) verbs. Thus, for example, if a FORTRAN routine writes to unit 6 and a COBOL85 routine executes a DISPLAY verb, both routines write to the same open of a Guardian file.

In the following example, a COBOL85 routine and a FORTRAN routine are bound into a single object file:

COBOL85 routine:

```
      DISPLAY "Hello from COBOL".
```

FORTRAN routine:

```
      OPEN (6, MODE = 'OUTPUT')
      WRITE (6, 100)
100   FORMAT(1X, 'Hello from FORTRAN')
```

Both the COBOL85 DISPLAY verb and the FORTRAN WRITE statement write to the same file open of the same Guardian file.

Shared access to units 5 and 6 enables you to coordinate FORTRAN I/O to unit 5 and unit 6 with I/O statements executed in modules written in languages other than FORTRAN. For example, if you run a program in which routines written in both C and in FORTRAN connect unit 5 to a disk file, alternate reads of the disk file by routines written in C and routines written in FORTRAN access successive records from the disk file, although both the C routines and the FORTRAN routines open the file in their own environment. Without file sharing, successive reads by routines written in C and in FORTRAN would repeat records already read by a routine in the other language. (Actually, without explicit support for file sharing, it is unlikely that routines written in C and in FORTRAN could be bound into one object file and successfully execute I/O statements.)

Note that only units 5 and 6 are shared with other routines and only if all routines specify ENV COMMON or default to a mode that Binder treats in the same class as ENV COMMON. Whether you specify ENV OLD or ENV COMMON, except for the files associated with unit 5 and unit 6, routines in two or more languages can access the same file with separate file opens. Each opener reads in succession, each record in the file, independent of other openers. (If one of the openers has the file open with protected access, the other opener, accessing the file with shared access, might be affected by the records written by the opener with protected access.)

In addition, whether you specify ENV OLD or ENV COMMON, you can share a file open between modules of a program by passing the Guardian file number.

For more information on how FORTRAN shares access to units 5 and 6—in particular, the values of the file attributes required to share file opens—see the OPEN Statement on page 7-70. For a detailed explanation of file sharing, see the *CRE Programmer's Guide*.

Module Compatibility

The Binder program defines three groups—or classes—of object files. The three Binder groups are old, common, and neutral. FORTRAN modules created with ENV OLD in effect are classified in the old Binder group. FORTRAN modules created with ENV COMMON in effect are classified in the common Binder group. You cannot create a FORTRAN module for the neutral Binder group. You can create neutral modules only in TAL and in Pascal.

When you create a new object file using Binder, the input files must all be in the

- Old or neutral groups and the resultant object file uses the a C-series FORTRAN run-time library.
- Common or neutral groups and the resultant object file uses the D-series FORTRAN run-time library.

You cannot bind together modules from both the old and the common groups.

For further information about Binder groups, see the *Binder Manual*.

Referencing Separately-Compiled Procedures

Your FORTRAN program can call routines that are compiled in separate compilations, including the ‘main’ procedure. For example, you might write your own FORTRAN subroutine to provide the parameters to a standard application.

Your FORTRAN routines can access the separately-compiled code using:

- The SEARCH directive to specify one or more disk-resident object files that contain object code created by the C, COBOL85, FORTRAN, Pascal, or TAL compilers.
- Binder to link program units written in different languages.
- Program libraries that contain compiled program modules for use by any program.

See the SEARCH Compiler Directive on page 10-61. Section 9, Program Compilation, includes an example of an independent compilation using the SEARCH directive.

Using Binder

You can enter Binder commands interactively or by placing the commands in a file that Binder reads. Binder creates an object file that includes the code blocks and data blocks you specify.

The following example shows an interactive Binder session in which code and data blocks from the object files TALPROC (a TAL file), COBPROC (a COBOL85 file), and FORTPROC (a FORTRAN file) are included in the object file TARGET (the @ is the Binder prompt):

```
1> BIND
@ADD * FROM cobproc
@ADD * FROM talproc
@ADD * FROM fortproc
@BUILD target
```

The ADD * command adds all the code and data blocks from an object file to be included in the object file created by Binder.

The BUILD command creates the new object file.

For additional information about Binder, see the *Binder Manual*.

Using Program Libraries

You can compile program modules and keep them in object module libraries for use by any program. FORTRAN programs can invoke library routines, including those written in languages other than FORTRAN.

Data Areas in User Library Space

FORTRAN routines in user libraries cannot directly access data items declared in:

- COMMON statements
- DATA statements
- SAVE statements
- The extended data segment

Library routines can reference data in the preceding data areas only if the calling routine passes the data item as an actual parameter to the library routine.

I/O Statements in User Library Space

Routines in library space can perform I/O operations but the units they access must be defined in user code space. Units referenced by routines in FORTRAN libraries must be defined in routines in user code space by one of the following:

- A UNIT compiler directive, such as:

```
?UNIT 8
```

- An executable I/O statement in user code space, not in user library space, that specifies a constant unit number, such as:

```
OPEN(8, FN = 'FT008')
```

The expression you specify for a unit number in an I/O statement in library code space must evaluate to a unit number defined in user code space. You must either pass the unit number to the library routine as an actual parameter or establish a convention for your application such that the library routine uses correct unit numbers.

Fault-Tolerant Statements in User Library Space

Routines in library space can execute START BACKUP and CHECKPOINT statements.

However, unit numbers must be defined in a routine in user code space using the same features as described in I/O Statements in User Library Space on page 13-5.

Data items named in START BACKUP and CHECKPOINT statements must conform to the rules stated in Data Areas in User Library Space on page 13-5.

Creating a User Library Space

To create a user library object file, use the same Binder commands—ADD, BUILD, and so forth—described in Using Binder on page 13-4.

Using Global Data in Mixed Language Programming

FORTRAN does not have global data in the same sense as some other programming languages such as C, Pascal, and TAL. Instead, FORTRAN programs share data by placing specific data items in common blocks and declaring those common blocks in each program unit that uses them.

FORTRAN follows the standard HP convention by adding a character at the beginning of the name of each COMMON block when it creates the corresponding data block in the object file. FORTRAN adds a period (.) if the block is in the user data segment, or a dollar sign ($) if the block is in the extended data segment. Blank common is called .BLANK^ if it is in the user data segment, $BLANK^ if it is in the extended data segment.

FORTRAN does not create a pointer block in the global primary data area for each COMMON data block. Instead, FORTRAN creates one “special” data block named COMMON#POINTERS in the global primary data area, which contains pointers to variables in all blocks in the user data segment. The contents of COMMON#POINTERS can be affected by the EXTENDCOMMON compiler directive. For more information, see Section 10, Compiler Directives. FORTRAN creates a pointer in each program unit’s local data area to each variable in the extended memory segment that is referenced in that program unit.

For example, if a FORTRAN source program declares:

```
? LARGECOMMON big
   COMMON /big/ a (100, 100), b (100, 100)
   COMMON /small/ c (10, 10), d, e
```

the FORTRAN object file contains:

- Common data block “$BIG” in the extended data segment
- Common data block “.SMALL” in the user data segment
- Special data block “COMMON#POINTERS” allocated in the primary data area and containing pointers to the variables your program references in data block “.SMALL”.

If a TAL source program declares:

```
BLOCK big;
   REAL .EXT a [1:10000], .EXT b [1:10000];
END BLOCK;

BLOCK small;
   REAL .c [1: 100], .d [0: 0], .e [0: 0];
END BLOCK;
```

the TAL object file includes:

- Common data block “$BIG” in the extended data segment
- Common data block “BIG” in the global primary data area, containing doubleword pointers to A and B
- Common data block “.SMALL” in the user data segment
- Common data block “SMALL” in the global primary data area, containing pointers to C, D, and E

The layout of blocks “$BIG” and “.SMALL” will be the same as those of the FORTRAN program.

When Binder combines these two object files, procedures in each object file can reference the variables in the two common data blocks safely, because they have equivalent declarations.

Note that each FORTRAN simple variable in a common block must be declared in TAL as a one-element array, so that TAL creates a pointer in the pointer block and the variable itself in the data block that corresponds to the FORTRAN common block, in order to achieve the desired equivalency of declarations.

The FORTRAN Calling Sequence

This subsection explains how FORTRAN generates object code to invoke subprograms that are not declared in GUARDIAN or CONSULT compiler directives and, thus, are assumed to be written in FORTRAN. The caller of a FORTRAN subprogram must set up the stack according to the conventions used by FORTRAN, regardless of the language in which the caller is written.

If you do not specify a GUARDIAN or CONSULT directive, FORTRAN does not know the calling sequence of the subprogram. It determines the calling sequence based on the actual arguments you pass to the routine. That is, the FORTRAN compiler cannot examine an object file containing the called procedure, or a source file containing its declaration.

Passing Parameters

Suppose a FORTRAN program includes the type declarations:

```
INTEGER*2 holmes, watson
REAL doyle
CHARACTER*6 conan
CHARACTER*7 mycroft
CHARACTER*8 sherlock
CHARACTER*9 moriarity
```

When FORTRAN translates the CALL statement

```
CALL bakerstreet (holmes, doyle)
```

it generates object code to call a procedure named BAKERSTREET which, if it were written in FORTRAN, would be a subroutine subprogram having formal parameters with types as shown. You could write the called routine in TAL as follows:

```
PROC bakerstreet (holmes, doyle);
   INT .holmes;
   REAL .doyle;
BEGIN
   ...
END;
```

If you specify the FORTRAN compiler directive EXTENDEDREF, you would code the TAL routine as follows:

```
PROC bakerstreet (holmes, doyle);
   INT .EXT holmes;
   REAL .EXT doyle;
BEGIN
   ...
END;
```

The calling sequence of a FORTRAN subroutine is the same as that of a TAL procedure with all parameters passed by reference. The data types of the caller's actual arguments must correspond to the data types of the called routine's dummy arguments. (For information on corresponding data types between different programming languages, see Appendix D, Data Type Correspondence and Return Value Sizes.) Reference parameters have word or doubleword addresses, depending on the value you specify for the EXTENDEDREF directive.

Similarly, FORTRAN translates the function reference:

```
diary = watson (holmes, doyle)
```

by generating object code to call a function procedure named WATSON which, if it were written in FORTRAN, would be a function subprogram having formal parameters

with types as shown and return a type INTEGER*2 function value. If it were written in TAL, it would be declared as follows:

```
INT PROC watson (holmes, doyle);
   INT .holmes;
   REAL .doyle;
BEGIN
   INT value;
   ...
   RETURN value;
END;
```

If you specify the FORTRAN compiler directive EXTENDEDREF, the TAL code would be:

```
INT PROC watson (holmes, doyle);
   INT .EXT holmes;
   REAL .EXT doyle;
BEGIN
   INT value;
   ...
   RETURN value;
END;
```

Calling a FORTRAN function is the same as calling a typed TAL procedure with the corresponding function value data type and with all parameters passed by reference.

Character Functions

FORTRAN treats a type CHARACTER function as a typeless procedure with an additional parameter, in which the function value is returned, preceding the arguments that appear in the FORTRAN source code. However, you must not declare this pseudo-parameter as a formal parameter if the procedure is coded in TAL. Thus, for example, given the function reference:

```
conan = mycroft (holmes, doyle)
```

FORTRAN generates object code to call a procedure that could be declared in TAL as:

```
PROC mycroft (holmes, doyle);
   INT .holmes;
   REAL .doyle;
BEGIN
   STRING .result = 'L' - 5;
   result ':=' ... ;
END;
```

Note that you do not specify a type in the TAL procedure for the TAL return value.

If you specify the FORTRAN compiler directive EXTENDEDREF:

```
PROC mycroft (holmes, doyle);
   INT .EXT holmes;
   REAL .EXT doyle;
BEGIN
   STRING .EXT result = 'L' - 8;
   result ':=' ... ;
END;
```

For the statement:

```
conan = mycroft (holmes, doyle)
```

FORTRAN generates code equivalent to:

```
STACK @temp
CALL mycroft (holmes, doyle)
conan = temp
```

where TEMP is a CHARACTER*7 variable created by the compiler, so that the function reference has the desired effect. This elicits a “return type mismatch” warning message from Binder, because FORTRAN describes the procedure to Binder as a function that returns a character string value, but the TAL code does not specify a return value.

Character Parameter Lengths

Whenever a subprogram has a type CHARACTER parameter, either as an explicit parameter or as the function-value pseudo-parameter for a CHARACTER function, FORTRAN stores a 16-bit word containing that parameter’s length on top of the stack, just below the words containing the parameter list.

If the procedure is not EXTENSIBLE, the length words are stored in the same relative order as the procedure’s type CHARACTER parameters, with no gaps for parameters

of types other than CHARACTER. For an EXTENSIBLE procedure, the length words are stored in the reverse order, so that you can add formal parameters of type CHARACTER to existing procedures.

FORTRAN passes the length words so that the called procedure knows the lengths of the actual arguments that correspond to the formal arguments, as well as the function return value pseudo-parameter, as type CHARACTER*(*). For example, given the FORTRAN statement:

```
conan = sherlock (holmes, moriarity, doyle)
```

the FORTRAN compiler generates object code equivalent to the following TAL source code:

```
STACK 8;              ! The length of SHERLOCK
STACK 9;              ! The length of MORIARITY
STACK @temp;          ! The function return address
CODE (PUSH %722);     ! Push the lengths and address into mem
CALL sherlock (holmes, moriarity, doyle);
CODE (ADDS -3);       ! Delete the non-parameter words
```

and follows this with object code for the FORTRAN statement:

```
conan = temp
```

where TEMP is a type CHARACTER*8 temporary variable created by the compiler for the function value returned by SHERLOCK. If you write the called function in TAL, you can reference the length words by declaring them equivalent to 'L' minus the appropriate number of words. For example, assuming you use EXTENDEDREF, you can code the function SHERLOCK in TAL as:

```
PROC sherlock (holmes, moriarity, doyle);
   INT .EXT holmes;
   STRING .EXT moriarity;
   REAL .EXT doyle;
BEGIN
   INT len_result = 'L' - 12;
   INT len_moriarity = 'L' - 11;
   STRING .EXT result = 'L' - 10;
   ...
   result ':=' ... FOR len_result;
END;
```

because the top few words of the stack will be as follows:

```
'L' - 0          Caller's (L)
'L' - 1          Caller's (E)
'L' - 2          Caller's (P)
'L' - 3, 4       Extended address of DOYLE
'L' - 5, 6       Extended address of MORIARITY
'L' - 7, 8       Extended address of HOLMES
'L' - 9,10       Extended address of RESULT
'L' - 11         Length of MORIARITY
'L' - 12         Length of RESULT
```

Calling Other Language Procedures From FORTRAN

If you write a FORTRAN program that calls subprograms written in other languages, you can use:

- The GUARDIAN directive for Guardian procedures and utility routines
- The CONSULT directive for user-supplied procedures written in C, COBOL85, FORTRAN, Pascal, and TAL

to give the FORTRAN compiler the information it needs so that you can invoke such routines from your FORTRAN program in the same way that you would call a FORTRAN subroutine or function subprogram. The compiler translates the argument list into an appropriate calling sequence according to the declaration of the referenced procedure, without special coding on your part.

To use the GUARDIAN and CONSULT directives you must have an object file from which the FORTRAN compiler can get the information about the called procedures.

The procedures in the file can be stubs (with empty or incomplete executable portions), because the FORTRAN compiler needs only the procedure’s name and attributes and attributes of each of its parameters.

The files required for the GUARDIAN directive are provided by HP along with the FORTRAN compiler, but you must create any files referenced by CONSULT directives. This means that procedures must exist (at least as stubs) before you can compile a FORTRAN program that refers to them with CONSULT directives.

You can also call subprograms written in another language from a FORTRAN program without using GUARDIAN or CONSULT directives, but you must code the procedure header to expect the default calling sequences that FORTRAN generates without these directives, as described in the preceding subsection.

General Restrictions

You cannot call a subprogram from FORTRAN if it:

- Has more than 63 formal parameters
- Has pass-by-value parameters larger than 64 bits
- Is a function that returns a value that is not a simple scalar value or is of a data type that cannot be declared in FORTRAN (for example, pointers and structures are not allowed)

You cannot call a function subprogram from FORTRAN with a CALL statement.

Using GUARDIAN and CONSULT Directives

To call a procedure declared in a GUARDIAN directive, find the description of the procedure you want to call in the *Guardian Procedure Calls Reference Manual* or for more information, see Section 15, Utility Routines. To call a procedure declared in a CONSULT directive, find the procedure's description in the documentation for the file named in the CONSULT directive.

Use the description to determine:

- Whether to call the procedure as a subroutine or as a function
- The order of its parameters
- Which parameters are required
- The data type of each parameter that you use
- Whether each argument that you use is passed by value or by reference

If the procedure returns a value, you must call it as a function from FORTRAN. Construct the call as a FORTRAN function reference. If you cannot call the procedure as a function, construct the call as a FORTRAN subroutine call, with a CALL statement. As in normal FORTRAN calls, arguments and parameters must match in order and type. Each argument that you supply must be a FORTRAN data type that corresponds to the dummy argument's data type. Appendix D describes the correspondence between data types in C, COBOL85, FORTRAN, Pascal, and TAL.

Unlike normal FORTRAN calls, you can omit optional arguments when calling procedures declared by GUARDIAN and CONSULT directives. To omit arguments at the end of the argument list, simply leave them off. To omit an argument from the middle of the argument list, omit the argument itself, but include the comma that would normally follow that argument.

Examples of Guardian Calls

1. This example calls the FILEERROR procedure to obtain status information about a file:

   ```
   istatus = FILEERROR (ifilenum)
   ```

2. This example calls the FILEINFO procedure to determine the number of extents allocated to a file:

   ```
   CALL FILEINFO (ifilenum,,, ,,, ,,, ,,, ,,, ,inumexts)
   ```

   The FILEINFO procedure actually has 26 parameters but only one required parameter. In the example, IFILENUM is a required argument and INUMEXTS is an optional argument.

   The call skips 15 optional parameters by using commas to mark their positions in the argument list. (There are 16 commas in the argument list, one to follow the initial argument and 15 more to indicate the omitted arguments.)

   Nine additional optional arguments are omitted from the end of the argument list.

Calls in Programs With Extended Data Space

If your FORTRAN program uses extended data space (that is, the program includes EXTENDEDREF, LARGECOMMON, or LARGEDATA directives), you can use a data item that is a formal parameter or resides in extended memory as an argument in a procedure call if the dummy argument is passed by extended reference (doubleword address), but not if it is passed by standard reference (word address).

Some C-series Guardian procedures expect pass-by-reference arguments to have word addresses. Data items stored in extended data space and all formal parameters in programs compiled with EXTENDEDREF, LARGECOMMON, or LARGEDATA directives have doubleword addresses.

If you need to call a Guardian procedure that has a word, pass-by-reference argument, but the argument you are passing is in extended memory, you can use a word temporary variable when you call the Guardian procedure. Assign the value of the extended-memory variable to the temporary variable. Use the temporary variable as an argument to the procedure and then assign the value of the temporary variable to the original variable after the call completes. The following example illustrates this:

```
?LARGEDATA error
?GUARDIAN fileinfo
   integer error, error_temp
   CALL fileinfo(5, error_temp)
   error = error_temp
```

The ERROR parameter to the FILEINFO procedure is a word, pass-by-reference parameter. Because the example specifies that ERROR is in extended memory

(LARGEDATA ERROR), you cannot pass its address to the FILEINFO procedure. The example passes ERROR_TEMP, which is allocated in the user data segment, instead, and then assigns the value returned in ERROR_TEMP to ERROR.

The *Guardian Procedure Calls Reference Manual* indicates which procedures expect word addresses and which expect doubleword addresses. Parameters that specify .EXT in their description expect doubleword addresses.

This restriction does not apply to arguments passed by value. You can use formal parameters and data items stored in extended memory as pass-by-value arguments in Guardian procedure calls.

Calling Routines Without Using GUARDIAN and CONSULT Directives

You can call subprograms that are not written in FORTRAN from FORTRAN without declaring them in GUARDIAN or CONSULT directives, but this requires a different syntax that is more difficult to use and this method usually causes many “parameter mismatch” and related warning messages from the Binder.

Note. Calling routines that are not written in FORTRAN without specifying them in a GUARDIAN or CONSULT directive was an HP FORTRAN feature before either the GUARDIAN or the CONSULT directives was supported. New FORTRAN code—and existing code that you modify—should specify external routines using either the GUARDIAN directive or the CONSULT directive. Existing programs that use the old form for procedure calls that are not written in FORTRAN will continue to compile and execute correctly unless you modify the programs in certain ways. For details, see Compatibility With the Old Form of Procedure Calls Not Written in FORTRAN on page 13-22.

If you write a FORTRAN program that calls a procedure that are not written in FORTRAN and you do not specify a GUARDIAN or CONSULT directive, you must observe the following rules:

- A procedure that can be called as a function must be called as a function from FORTRAN.
- All parameters are required, even if you call a VARIABLE or EXTENSIBLE procedure. See below for more information on optional parameters.
- The order of all parameters must match.
- The number of parameter words passed by FORTRAN must match the number of words expected by the called procedure.

  In general, this means that the parameters must match in number and type, and that you must supply parameters of a FORTRAN type that matches the type expected by the called procedure. For example, if the called procedure is written in TAL and expects a parameter that is declared INT(32), you must supply a FORTRAN argument that is declared INTEGER*4.

- Arguments passed by reference must pass word or doubleword addresses, according to the requirements of the called procedure. All parameters passed by reference must have the same size address, either word or doubleword. If the called procedure expects word addresses, the FORTRAN program must not be compiled with any EXTENDEDREF, LARGECOMMON, or LARGEDATA compiler directives. If the called procedure expects doubleword addresses, the FORTRAN program must be compiled with at least one of these compiler directives.

  For example, a TAL procedure that begins

  ```
  PROC flipout (input, output);
     INT .input;
     INT .EXT output;
  ```

  cannot be called from a FORTRAN program, because the first parameter, INPUT, requires a word address and the second parameter, OUTPUT, requires a doubleword address, but a single FORTRAN program cannot have both word and doubleword addresses unless the procedure is declared with a GUARDIAN or CONSULT directive.

- Parameters passed by value must be surrounded by backslash characters in the argument list, as in the following example:

  ```
  CALL procedurename ( ... , \X\, ... )
  ```

  You cannot pass arrays nor can you pass type character values by value. Using backslash characters around value parameters is an HP extension to the ANSI FORTRAN language standard.

- If a procedure has an optional parameter that you want to omit in a particular call, you must include a “dummy” value to serve as a placeholder in the argument list. If the parameter is passed by value, you can supply a dummy argument like this:

  ```
  CALL procedurename ( ... , \0\, ... )
  ```

  If you want to omit a value for a multiple-word optional parameter that is passed by value, you must supply multiple dummy values to serve as placeholders for each word that corresponds to that parameter. For example, you can omit a doubleword, optional, pass-by-value parameter like this:

  ```
  CALL procedurename ( ... , \0\, \0\, ... )
  ```

  In this case, you must violate the rule “parameters must match in number” in order to fulfill the rule “parameters must match in type.”

- If a procedure has optional parameters, you must also supply a “mask” parameter as an extra, final parameter, which does not appear in the formal parameter list in the procedure’s declaration in its source language. The exact format of the mask depends on whether the procedure is declared VARIABLE or EXTENSIBLE.

  Masks for VARIABLE procedures:

- The mask must be one word for procedures with up to 16 parameters and two words for procedures with 17 to 29 parameters.
- The mask value must be right-justified in the word or words.
- The bits in the mask correspond to parameters: bit 15 of the last word in the mask is the bit associated with the last parameter; other bits correspond to the remaining parameters in order.
- Set the bit for each parameter whose value you supply in the call. Don't set bits for optional parameters that you omit, although you supply "dummy" values as placeholders for them.

Masks for EXTENSIBLE procedures:

- The mask must be one word for each 16 words of parameters or portion thereof. For example, if there are 23 parameters some of which are multi-word parameters (extended reference parameters or 32-bit or 64-bit value parameters) so that there are a total of 37 words of parameters, you must have three words of mask bits.
- The mask value must be left-justified in the word or words.
- The bits correspond to words of parameters: bit 0 of the first word in the mask is the bit associated with the first word of the first parameter; other bits correspond to the remaining parameter words in order.
- Set the bit for each word of each parameter whose value you supply in the call. Don't set bits for words of optional parameters that you omit, although you supply "dummy" values as placeholders for them.
- After the last mask word, you must include a single word that contains the negative of the number of words of parameters passed (not counting the mask words), as a 16-bit twos-complement binary integer.

If an EXTENSIBLE procedure is declared in TAL with a number in parentheses after the word EXTENSIBLE, you can also call the procedure as a VARIABLE procedure. If you do this, however, you can only use the first n parameters, where n is the number in parentheses. Build the procedure call exactly as if the procedure was declared VARIABLE and ignore all the additional parameters. In effect, you are using an older version of the procedure that has fewer parameters.

Calling TAL Subprograms From FORTRAN

If you write a FORTRAN program that calls a TAL procedure, you must observe the following rules:

- For programs that you compile with ENV OLD in effect, the main program must be written in FORTRAN, so that it establishes the run-time environment required by FORTRAN object code.
- You can do I/O in TAL and in FORTRAN within the same program.

If you compile your FORTRAN modules with ENV OLD in effect and your TAL modules with either ENV OLD or ENV NEUTRAL in effect, routines written in both languages can open the same file but each open is independent of all other opens.

You can share access to the same file open by passing the file number between routines that need to access the same file using the same file open.

If you compile your FORTRAN modules with ENV COMMON and your TAL modules with either ENV COMMON or ENV NEUTRAL, your FORTRAN and TAL routines can share access to the same Guardian file open for standard input and for standard output—the files associated with unit 5 and unit 6, respectively. To share standard input or standard output, a routine in each language must explicitly open the file.

The TAL procedures can use the Guardian file system, the Sequential Input/Output (SIO) package of GPLIB, or embedded NonStop SQL statements without interfering with FORTRAN I/O statements in the same program.

For more information about shared files, see the OPEN Statement on page 7-70.

- If the FORTRAN program does not declare the TAL procedure with a GUARDIAN or CONSULT directive:
  - If the FORTRAN program is compiled with any EXTENDEDREF, LARGECOMMON, or LARGEDATA directives, the called TAL procedure must declare all reference parameters (if any) with .EXT, so the TAL procedure will expect the doubleword argument addresses that FORTRAN provides.
  - Otherwise, the called TAL procedure must declare all reference parameters (if any) without .EXT, so the TAL procedure will expect the word-address arguments that FORTRAN provides in this case.
- For a TAL type UNSIGNED(8) formal parameter passed by value, FORTRAN allows a CHARACTER*1 expression or an integer constant with value in the range 0 through 255 as the argument.
- For a TAL type UNSIGNED(16) formal parameter passed by value, FORTRAN allows an integer constant with value in the range 0 through 65,535 as the argument.
- For a TAL type UNSIGNED(31) formal parameter passed by value, FORTRAN allows an integer constant with value in the range 0 through 3,147,483,647 as the argument.
- If a called TAL procedure has parameter-pair formal parameters, FORTRAN generates an address-length pair actual parameter on the stack according to the

TAL convention, and does not stack length words for character parameters. For example, if the TAL procedure is declared by:

```
PROC p (s: l);
   STRING .EXT s;
   INT l;
EXTERNAL;
```

the colon between S and L causes the TAL compiler to declare (to the Binder) that L is a parameter pair. The corresponding references to this procedure P in the following FORTRAN and TAL subprograms are equivalent:

```
                          PROC tal;
SUBROUTINE fortran        BEGIN
CHARACTER * 20 c            STRING .c[0: 19];
CALL p(c)                  CALL p( c:20);
CALL p(c (6: 12))          CALL p( c[5]:7 );
END                       END;
```

In each subprogram the first CALL statement passes all of C with a length of twenty characters, and the second CALL statement passes a substring consisting of the sixth through twelfth characters of C with a length of seven characters. Note that each CALL statement in the FORTRAN routine has only one argument. The length value argument is implicit.

Calling COBOL85 Subprograms From FORTRAN

If you write a FORTRAN subprogram that calls a subprogram written in COBOL85, you must observe the following rules:

- If your COBOL85 and FORTRAN programs are compiled with ENV OLD in effect, the main program must be written in COBOL85, so that it establishes the run-time environment required by COBOL85 object code. FORTRAN's requirements are a subset of COBOL85's. This restriction does not apply if you specify ENV COMMON.
- Regardless of whether you compile with ENV OLD or ENV COMMON, you can do I/O in COBOL85 and FORTRAN in the same program but not, in general, on the same file.

If you specify ENV COMMON, your FORTRAN and COBOL85 routines can share the same file open to the standard input and standard output files—unit 5 and unit 6 in FORTRAN. In COBOL85 you reference the standard input file when you execute an ACCEPT verb; you reference standard output when you execute a DISPLAY verb. See Using Shared Files on page 13-27.

- If the FORTRAN program does not declare the COBOL85 subprogram with a CONSULT directive:

  - If the FORTRAN subprogram is compiled with any EXTENDEDREF, LARGECOMMON, or LARGEDATA directives, the ACCESS MODE IS EXTENDED-STORAGE clause must be included in each level 01 or 77 data item in the Linkage Section of the Data Division of the called COBOL85 subprogram, so that the called COBOL85 subprogram will expect the doubleword argument addresses that FORTRAN will provide.
  - Otherwise, the ACCESS MODE IS STANDARD clause must be specified or assumed for each item in the Linkage Section, so that the called COBOL85 subprogram will expect the word-address arguments that FORTRAN generates in this case.

- The called COBOL85 subprogram's Procedure Division must begin with a header that includes the USING phrase if you want to pass parameters to the COBOL85 routine.
- Your FORTRAN program can call a COBOL85 subprogram that has no parameters. The COBOL85 program's Data Division must not have a Linkage Section, its Procedure Division header must not have a USING phrase, and the compilation unit must specify a COBOL85 MAIN directive. The name specified on the MAIN directive must be different than the name of the COBOL85 subprogram that you call. Here is an example:

```
?MAIN NotFortranProgram
   IDENTIFICATION DIVISION.
   PROGRAM-ID. FortranProgram.
   DATA DIVISION.
   ...
   PROCEDURE DIVISION.
   ...
```

- FORTRAN can call COBOL85 subprograms only as subroutines not as functions.

Calling C Subprograms From FORTRAN

If you write a FORTRAN program that calls a C subprogram (that is, a C function), you must observe the following rules:

- You cannot bind FORTRAN routines compiled with ENV OLD in effect with C subprograms.
- The name of the C subprogram must not contain any lower case letters, because FORTRAN always upshifts all names.
- The C subprogram must not be compiled with the OLDCALLS pragma.
- The prototype of the C subprogram must not specify a variable number of parameters.

- If the FORTRAN program does not declare the C subprogram with a GUARDIAN or CONSULT directive:
  - If the FORTRAN program is compiled with any EXTENDEDREF, LARGECOMMON, or LARGEDATA directives, the called C subprogram must be compiled with the XMEM directive, so that the C subprogram will expect the doubleword-address arguments that FORTRAN passes.
  - Otherwise, the called C subprogram must be compiled with the NOXMEM directive, so that the C subprogram will expect the word-address arguments that FORTRAN generates in this case.
- If the C subprogram's function type is "void," it must be invoked by a CALL statement in FORTRAN; otherwise, it must be invoked by a function reference.
- If the C subprogram's function type is "char" or "unsigned char," FORTRAN considers the function to be a CHARACTER*1 function.
- For a C type "char" or "unsigned char" formal parameter passed by value, FORTRAN allows a CHARACTER*1 expression or an integer constant with value in the range 0 through 255 as the actual argument.
- For a C type "unsigned int" formal parameter passed by value, FORTRAN allows an integer constant with value in the range 0 through 65,535 as the argument.
- For a C type "unsigned long int" formal parameter passed by value, FORTRAN allows an integer constant with value in the range 0 through 4,294,967,295 as the argument.

Calling Pascal Subprograms From FORTRAN

If you write a FORTRAN program that calls a Pascal subprogram (that is, a Pascal function or procedure), you must observe the following rules:

- You cannot bind FORTRAN routines compiled with ENV OLD in effect with Pascal subprograms.
- If the FORTRAN program does not declare the Pascal subprogram with a GUARDIAN or CONSULT directive:
  - If the FORTRAN program is compiled with any EXTENDEDREF, LARGECOMMON, or LARGEDATA directives, the called Pascal subprogram must be compiled with the XMEM directive, so the Pascal subprogram will expect the doubleword argument addresses that FORTRAN passes.
  - Otherwise, the called Pascal subprogram must be compiled with the NOXMEM directive, so the Pascal procedure will expect the word-address arguments that FORTRAN passes in this case.
- If you call a Pascal subprogram that is EXTENSIBLE, FORTRAN cannot tell which parameters are declared OPTIONAL, and assumes that all arguments are optional.

The COBOLEXT Files

Every NonStop system in which the FORTRAN compiler is installed, also includes files named COBOLEX0, COBOLEX1, and COBOLEXT in the same subvolume as the FORTRAN compiler (usually this is the $SYSTEM.SYSTEM subvolume). These files are also used by the COBOL85 compiler.

The compilers use these files when translating references to Guardian procedures. They are object files and contain, in effect, compiled versions of the source files EXTDECS0, EXTDECS1, and EXTDECS, respectively, that are also available in the $SYSTEM.SYSTEM subvolume of every NonStop system.

If the FORTRAN compiler encounters a CALL statement or function reference to a procedure that was named in a GUARDIAN directive, the compiler consults the file named COBOLEXT in the same subvolume as the compiler for information about the procedure and its parameters, so that the compiler can generate the correct object code for invoking the procedure. The file named COBOLEXT corresponds to the EXTDECS file, which contains declarations of Guardian procedures as of two major releases preceding the current release. If you want to use a more recent version such as COBOLEX1 or COBOLEX0 (such as using a Guardian procedure that was added in the latest or previous major release), you must use the CONSULT directive, rather than the GUARDIAN directive, to reference one of those files by name.

Compatibility With the Old Form of Procedure Calls Not Written in FORTRAN

Existing programs that use the old form for procedure calls that are not written in FORTRAN will continue to compile and execute correctly unless you modify the programs to use extended data space (that is, to use EXTENDEDREF, LARGECOMMON, or LARGEDATA directives), or unless you declare the procedures that are not written in FORTRAN using GUARDIAN and CONSULT directives.

- Because the new calling sequence is easier to code and read, you should add GUARDIAN and CONSULT directives and update the calling sequences if you modify existing programs. It is not essential that you do this, however, except in a few specific cases:
- You must add GUARDIAN and CONSULT directives and update the calling sequences if you modify the programs to use EXTENDEDREF, LARGECOMMON, or LARGEDATA directives.

You must use the new calling sequence for any procedure not written in FORTRAN that you declare in a GUARDIAN or CONSULT directive, even if the program does not use EXTENDEDREF, LARGECOMMON, or LARGEDATA directives.

To convert existing procedure calls that are not written in FORTRAN to the new form for procedure calls not written in FORTRAN, follow these steps:

1. Delete any variable or extensible mask word (or words) at the end of the argument list.

2. Delete dummy values used as placeholders for omitted arguments. If the omitted arguments are at the end of the argument list, delete them completely; if the omitted arguments precede other arguments that you supply in the list, use successive commas to skip them.
3. Delete the backslashes (\) that surround pass-by-value arguments.

You must make the changes listed in steps 1 and 2, but you do not have to remove the backslashes as indicated in step 3. FORTRAN generates warning messages if it finds backslashes surrounding pass-by-value arguments in calls to routines declared in procedures that are not written in FORTRAN, but it generates correct code in spite of the warning.

Calling FORTRAN Procedures From Other Languages

If you write a program in a language other than FORTRAN and this program calls a FORTRAN subprogram, you must observe the following rules:

- Declare the FORTRAN subprogram in the manner that the calling program's language requires.
- Write calls to it in such a way that parameter passing will work as described in The FORTRAN Calling Sequence on page 13-7, so that the FORTRAN subprogram receives the calling sequence that it expects.
- Don't call a FORTRAN type CHARACTER function from another language, because FORTRAN expects a zero-th parameter through which to return the function result value.

Calling FORTRAN Subprograms From TAL

If you write a TAL procedure that calls a FORTRAN subprogram, you must write TAL text to declare the FORTRAN subprogram as a TAL external procedure.

If the FORTRAN subprogram is compiled with an EXTENDEDREF, LARGECOMMON, or LARGEDATA directive, declare all the address parameters in the TAL external declaration as extended reference parameters—that is, declared with .EXT, as in the following example:

```
PROC fort_sub = "THE_SUB" (a, b, c) LANGUAGE FORTRAN;
   INT .EXT a,
      .EXT b,
      .EXT c;
EXTERNAL;
```

If the FORTRAN routine is not compiled with any of these directives, declare all the address parameters in the TAL external declaration as standard (that is, word) reference parameters (without .EXT) as in the following example:

```
PROC fort_sub = "THE_SUB" (a, b, c) LANGUAGE FORTRAN;
   INT .a,
      .b,
      .c;
EXTERNAL;
```

Precede the CALL with STACK and PUSH statements to place CHARACTER parameter lengths on the stack as expected by the FORTRAN subprogram, and follow the CALL with an ADDS instruction to delete those length words.

You can omit the CHARACTER parameter length words if you know that the FORTRAN subprogram does not declare them as assumed length dummy arguments with CHARACTER * (*) declarations.

Calling FORTRAN Subprograms From COBOL85

If you write a COBOL85 program that calls a FORTRAN subprogram, you must compile the FORTRAN subprogram first, because the COBOL85 compiler examines the object file that contains the compiled subprogram, and generates object code that sets up parameters in the way the called subprogram expects them.

The Procedure Division statement format is:

```
ENTER [ FORTRAN ] subprogram [ IN object-file ]
   [ USING parameter [ , parameter ] ... ]
   [ GIVING return-value ].
```

where *object-file* is a file-mnemonic defined in the SPECIAL-NAMES paragraph of the Environment Division. If you omit the IN (or OF) object-file phrase, the COBOL85 compiler searches the object files named in SEARCH directives, LIBRARY directives, and CONSULT directives, in that order.

The parameters listed in the USING phrase must agree in number, order, kind, data type, and dimensions with those of the called subprogram. For data type correspondence rules, see Appendix D, Data Type Correspondence and Return Value Sizes. If the called subprogram is a FORTRAN procedure, COBOL85 ensures that the FORTRAN run-time receives parameter addresses according to the declaration of your FORTRAN routine and CHARACTER parameter lengths as it expects them.

The GIVING phrase must be present if and only if the called subprogram is a function, and the data type of return-value must be compatible with that of the subprogram itself.

The called FORTRAN subprogram can be compiled with or without the EXTENDEDREF compiler directive. COBOL85 generates the correct word or

doubleword parameter addresses in either case, after examining the compiled procedure in its object file.

For information about sharing files in programs that consist of COBOL85 and FORTRAN routines, see Calling COBOL85 Subprograms From FORTRAN on page 13-19.

Calling FORTRAN Subprograms From C

If you write a C program that calls a FORTRAN subprogram, you must write C language text to declare (import) the FORTRAN subprogram as a C external function. Its C type is “void” if it is a FORTRAN subroutine.

If the FORTRAN subprogram is compiled with the EXTENDEDREF, LARGECOMMON, or LARGEDATA directive, declare all the C formal parameters as extended reference parameters (with XMEM). Otherwise, declare all of its formal parameters as standard reference parameters (with NOXMEM).

You cannot call a FORTRAN subprogram that has dummy arguments declared as CHARACTER * (*) , because C does not stack the required length words.

If you specify ENV OLD in your FORTRAN modules, you must use a C-series C compiler and you cannot do I/O in FORTRAN if the main routine is written in C.

Calling FORTRAN Subprograms From Pascal

If you write a Pascal program that calls a FORTRAN subprogram, you must write Pascal language text to declare the FORTRAN subprogram as a Pascal external function or procedure.

If the FORTRAN subprogram is compiled with the EXTENDEDREF, LARGECOMMON, or LARGEDATA directive, declare all of its formal parameters as extended reference parameters (with XMEM). Otherwise, declare all of its formal parameters as standard reference parameters (with NOXMEM).

You cannot call a FORTRAN subprogram that has any formal parameters declared as CHARACTER * (*) in FORTRAN, because Pascal does not stack the required length words.

If you specify ENV OLD in your FORTRAN modules, you must use a C-series Pascal compiler and you cannot do I/O in FORTRAN if the main routine is written in Pascal.

Intrinsic Function Declarations

NonStop systems on which the FORTRAN compiler is installed also include files named FORTDECS and FORTLIB, in the same subvolume as the FORTRAN compiler itself (usually this is the $SYSTEM.SYSTEM.* subvolume).

FORTDECS is an EDIT format file that contains TAL external procedure declarations for all the procedures that implement the intrinsic functions of the FORTRAN language, and FORTLIB is an object file that contains those procedures.

If you are writing a TAL program and you want to use any of the FORTRAN intrinsic functions, you can use TAL compiler directives to SOURCE in the FORTDECS file and SEARCH the FORTLIB file.

The FORTDECS file contains a SECTION for each procedure, so that you can use it in much the same way as the $SYSTEM.SYSTEM.EXTDECS file. The name of each procedure, and of its SECTION, is the FORTRAN intrinsic function name, with a circumflex character (^) appended, for word addressing of parameters. There is also a second version of each procedure, named by appending ^^EXT to the FORTRAN intrinsic function name, for doubleword extended addressing of parameters. For example, to use the SQRT function:

```
?SOURCE $SYSTEM.SYSTEM.FORTDECS (SQRT^)

?SEARCH $SYSTEM.SYSTEM.FORTLIB
```

Those intrinsic functions that perform validity checks on their arguments (see Intrinsic Errors on page G-3) contain references to the LIB^ERROR procedure, which resides in the same FORTLIB file.LIB^ERROR in turn calls proc FLIB^MESSAGE which normally resides in the system library.

Since TAL lacks a COMPLEX data type, COMPLEX values are declared as REAL(64) in the intrinsic function procedures. A COMPLEX data value is actually represented as a pair of REAL(32) values: the real part followed by the imaginary part.

Using ENV COMMON

The ENV compiler directive enables you to specify whether you want your FORTRAN program to use routines in the C-series or in the D-series FORTRAN run-time library.

If you specify ENV OLD or do not specify an ENV directive, your program uses routines in the C-series FORTRAN run-time library and cannot share files or other resources with other routines in your process. You do not need to read further in this subsection.

If you specify an ENV COMMON compiler directive, your program uses the routines in the D-series FORTRAN run-time library. The D-series FORTRAN run-time library enhances your program's ability to share resources by using routines in the Common Run-Time Environment (CRE). The FORTRAN run-time library depends on CRE routines to:

- Share access to the files standard input and standard output. Unit 5 references standard input. Unit 6 references standard output. If opened for shared access, the FORTRAN run-time library calls CRE routines whenever your program performs I/O on unit 5 or unit 6. For more information about shared access to standard input and standard output, see the OPEN Statement on page 7-70.
- Manage your process's backup process if your program runs as a NonStop process.
- Display diagnostic messages on the standard log file.

- Manage traps.
- Provide standard math routines that support FORTRAN intrinsic functions.
- Manage $RECEIVE.
- Ensure proper initialization and graceful termination of your program.

Using Shared Files

The CRE provides shared access to three files:

- Standard input
- Standard output
- Standard log

Sharing File Opens

A FORTRAN program accesses standard input if it reads from unit 5. It accesses standard output if it writes to unit 6. A FORTRAN program cannot establish a unit connection to standard log, although the FORTRAN run-time library can write messages to standard log, typically if errors occur while a FORTRAN module is executing. In addition, FORTRAN writes to standard log the message you specify on a PAUSE or STOP statement and in a call to the FORTRAN_COMPLETION_ utility.

Except for units 5 and 6, all FORTRAN units have the same semantics when you specify ENV COMMON as they do when you specify ENV OLD or you run a program compiled with a C-series FORTRAN compiler.

Using Unit 5 and Unit 6

If you open unit 5 or unit 6, either implicitly or without specifying the MODE specifier, your program accesses units 5 and 6 just as it does in C-series software. The default access MODE for both units is I-O. If you open unit 5 or unit 6 with MODE = I-O, FORTRAN establishes a connection between the specified unit and a Guardian file. Your program does not share a file open—a path to the file—with routines written in other languages and bound, along with your FORTRAN routines, into a single object file.

If you change the access MODE for unit 5 to 'INPUT' or for unit 6 to 'OUTPUT', the FORTRAN run-time library calls routines in the CRE to manage the standard input and standard output files. (Additional parameters can affect whether the FORTRAN run-time library calls CRE routines. For additional information, see the OPEN Statement on page 7-70.) In this case, your FORTRAN routines share access to a single Guardian open of standard input and a single Guardian open of standard output with routines in your process that are written in other languages. The CRE manages the single file open. Upon receiving its first request to open a shared file (that is, unit 5 or unit 6 for FORTRAN), the CRE opens the specified Guardian file. However, subsequent requests to open standard input or standard output from routines written in different

languages are managed by CRE routines. All such open requests to the same standard file access not only the same Guardian file, but access the file using the same file open.

For your program to share access to the standard files, it does not have to take any action beyond ensuring that it opens the standard file with the proper access MODE.

For more detailed information, see the *CRE Programmer's Guide*.

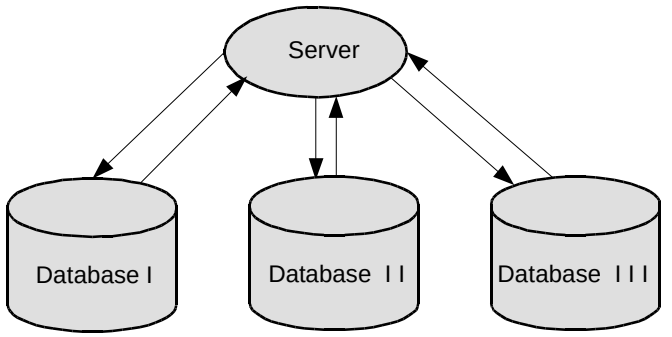
14 Interprocess Communication

Topics covered in this section include:

An application in an HP NonStop environment frequently consists of numerous communicating processes. Each process is designed to perform a particular set of tasks. Dividing an application into multiple processes enables you to more easily maintain your application because you design each process to perform a series of related functions. In addition, your application can take advantage of the multiprocessor hardware of HP NonStop systems by running processes in different CPUs, thereby achieving a high degree of parallelism. Finally, you can run identical processes in different CPUs and distribute activities among them, further increasing the amount of parallelism in your application.

For example, your application might have three separate databases. You might have a process that accesses databases for your application, rather than having each process in the application access the databases. In so doing, you further divide your application by isolating in one process all the code that accesses databases. Figure 14-1 illustrates this.

Figure 14-1. A Process That Access Databases



You might further refine your application by having a separate process for each of the databases in your application and isolate in each of these processes all the code required to access a single database. In addition, you might run identical processes in multiple CPUs. When a process in your application requires information from a particular database, it can choose from all the processes that access that database the process that is least busy. In Figure 14-2 on page 14-2, a process can access database 1 by sending a request to any of servers A, B, or C.

The processes described in the preceding paragraphs are called server processes. A server process accepts requests from other processes, carries out the requests received, and sends replies to the requesting processes.

A process that issues requests to a server process is called a requester process. A requester process is frequently referred to as a requester. A server process is frequently referred to as a server.

A server process provides services to requester processes. In the preceding examples, the server processes provide controlled access to databases. Servers can provide controlled access to any device or might provide services such as a computation without necessarily accessing any device.

Figure 14-2. Multiple Processes Accessing the Same Databases

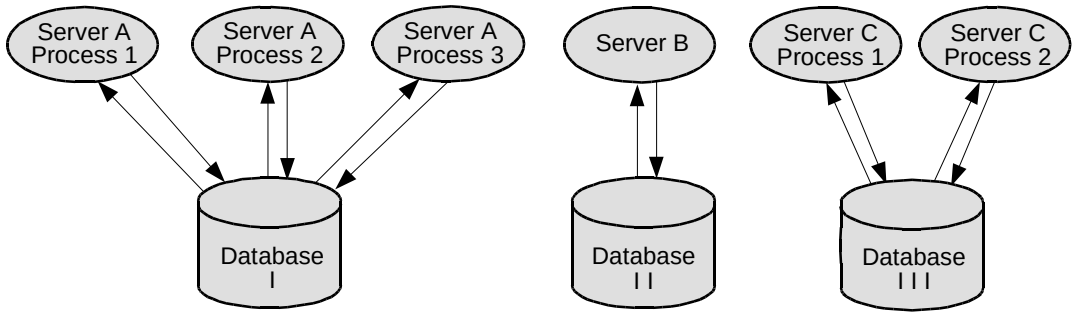
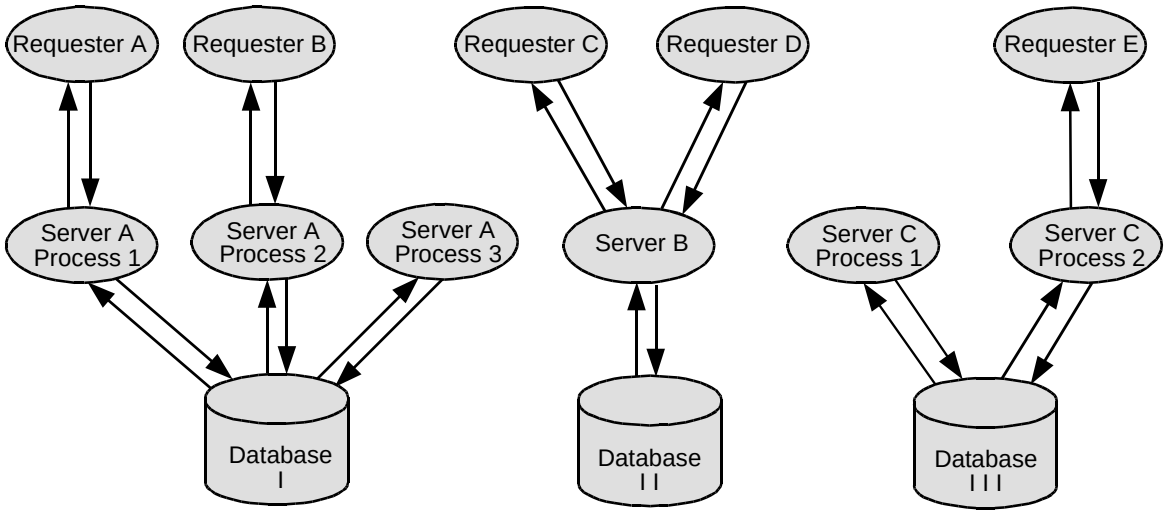


Figure 14-3 on page 14-3 shows multiple requesters and multiple servers that access three different databases. Requesters A and B access server processes A1 and A2 respectively. Server process A3 is not currently active. Requesters C and D are accessing server B. Requester E is accessing server C2. Server C1 is not currently active. When you design your application, you can specify whether you want a given server to accept more than one request at a time, as is the case with server B.

Figure 14-3. Requesters and Servers



Communication between processes takes place by means of a special file called $RECEIVE. For a comprehensive description of $RECEIVE, see the *Guardian Programmer's Guide*.

Managing $RECEIVE

The FORTRAN run-time environment maintains tables that enable it to manage $RECEIVE. It determines the size of the tables based on the values you give for the option specifiers to the RECEIVE directive.

The RECEIVE directive includes the following option specifiers:

- OPEN open

  open specifies the maximum number of opens from other processes that your server process can manage simultaneously. The system returns error 12, "File in Use," if it receives an open request that would exceed the number you specify for OPEN.

- SYNCDEPTH sync

  sync specifies the maximum number of replies that your program can hold for each requester in case the requester's backup process becomes its primary process and begins executing the statements that appear immediately after the requester's most recent CHECKPOINT statement. The value you specify for *sync* is application dependent. In general, *sync* specifies the maximum number of messages that any requester can send to your process before the requester executes a FORTRAN CHECKPOINT statement or calls a checkpoint system procedure such as CHECKPOINT, CHECKPOINTX, CHECKPOINTMANY, CHECKPOINTMANYX, and so forth.

The FORTRAN run-time library checks each message it receives from a requester process to determine if it has already received and replied to the message. If it has, it locates the previously-sent reply and returns it again. Your FORTRAN program is not aware of the retransmission.

- MAXREPLY *reply*

  reply specifies the maximum number of bytes in each reply message that your program must save.

- QDEPTH *depth*

  depth specifies the maximum number of messages your program can read from $RECEIVE without sending a reply to a unit that is connected to $RECEIVE.

- SYSMSG

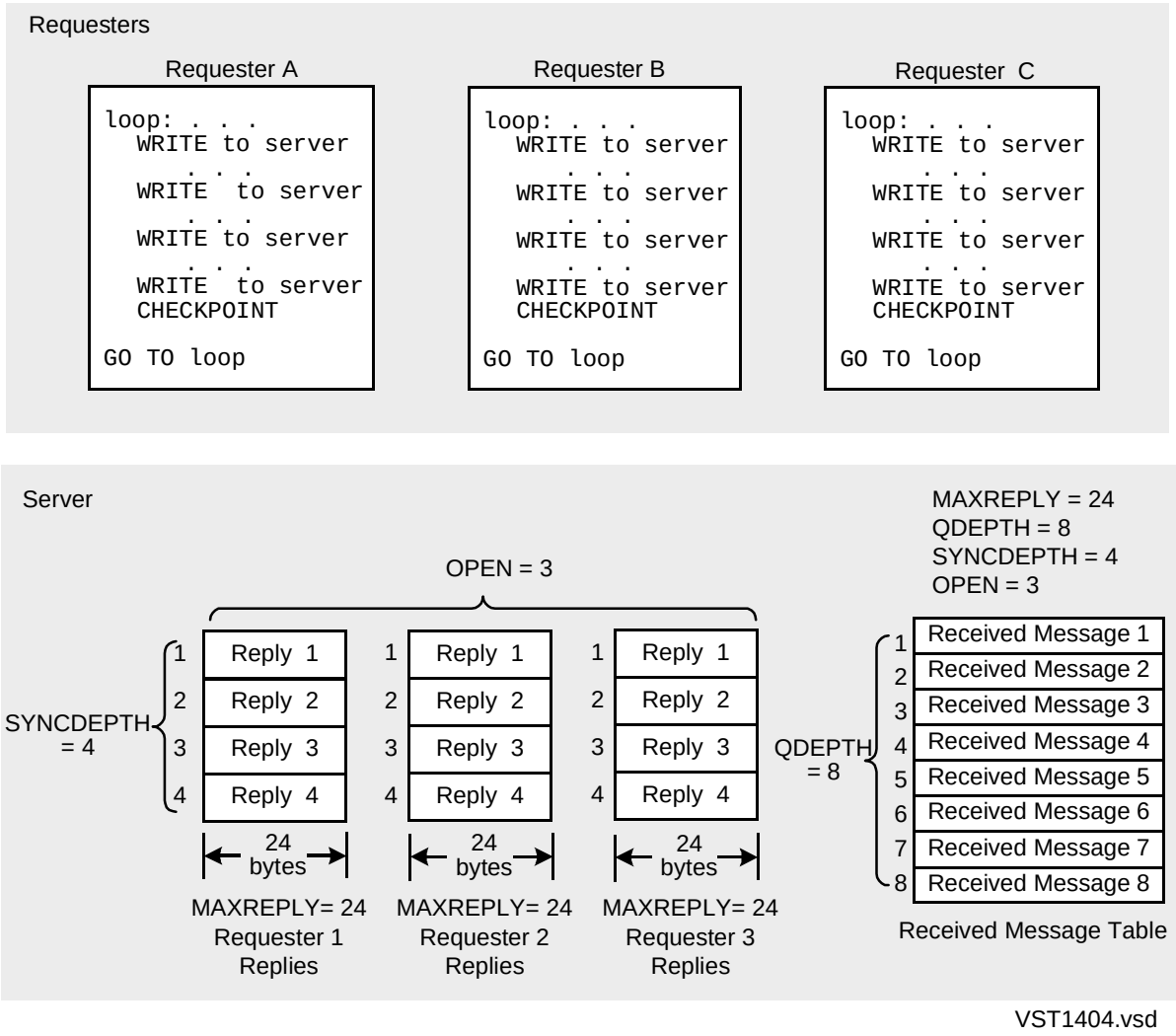
  If you specify SYSMSG, the FORTRAN run-time library delivers system messages (OPEN, CLOSE, SETMODE, BREAK, and so forth) received from $RECEIVE to your program. If you do not specify SYSMSG, the FORTRAN run-time library processes system messages.

  FORTRAN responds to SETMODE and CONTROL messages with file system error 2 if you do not specify SYSMSG. The unit receiving system messages via $RECEIVE must have a record length of at least 34 characters. For a description of system message formats, see the *Guardian Procedure Errors and Messages Manual* .

The following table explains how FORTRAN uses the structures shown in Figure 14-4 on page 14-5 based on each of the RECEIVE directive option specifiers:

Attribute	Effect
OPEN	The server can support three simultaneous NonStop requester processes because it has allocated space for three sets of server replies: Requester 1 Replies, Requester 2 Replies, and Requester 3 Replies. Therefore, the server can support all three requesters: Requester A, Requester B, and Requester C. It cannot, however, support a fourth requester.
MAXREPLY	The server can store replies that are up to 24 bytes long, as shown in each Reply table.
SYNCDEPTH	The server can support requesters that issue up to four requests to the server between calls to a CHECKPOINT statement or a checkpoint system procedure such as CHECKPOINT, CHECKPOINTX, CHECKPOINTMANYX, and so forth.
QDEPTH	The server's Received Message Table holds messages received from $RECEIVE for which a corresponding WRITE statement has not been executed.

Figure 14-4. Structure Allocation to Support NonStop Requester Processes



Using $RECEIVE

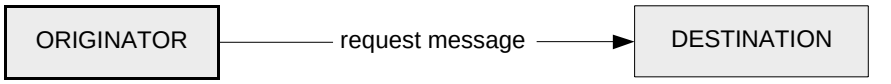
You can use $RECEIVE for interprocess communication in one of three ways: as an input file, as an input/output file, or as two separate files for independent input and output.

Observe the following when using $RECEIVE files:

- You can assign any FORTRAN unit to $RECEIVE through a run-time file assignment, a UNIT compiler directive, or an OPEN statement.
- You must access a unit connected to $RECEIVE as a sequential file.
- A write operation to a unit connected to $RECEIVE is valid only after a successful read operation from a (possibly different) unit assigned to $RECEIVE.

$RECEIVE as an Input File

In the simplest case, the task-handling process uses $RECEIVE as an input file only. It receives request messages from originating processes via $RECEIVE and acts upon the requests. It does not generate a reply to each message, but an automatic reply is sent when the next read operation is initiated. (Note that this default reply facility is available only if QDEPTH is equal to 1.)



VST1405.vsd

For example, a program might send print data to a spooler process instead of directly to a printer. This program, the originator, might contain the following statements:

```
        ...
   OPEN (UNIT=1,FILE='$spool')
        ...
   WRITE (UNIT=1,FMT=11) a,b,c
11 FORMAT(1X,3F9.4)
        ...
```

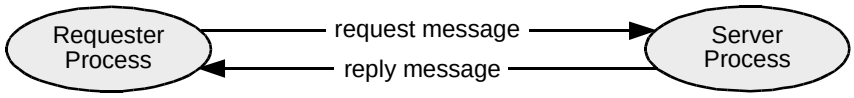
The spooler process, the destination, receives the print lines by reading $RECEIVE:

```
?RECEIVE (OPEN 2)
   CHARACTER*1 line(132)
   OPEN (UNIT=1,FILE='$RECEIVE')
        ...
10 READ (UNIT=1,END=100) line
   WRITE (UNIT=2) line
   GO TO 10
        ...
100 STOP
   END
```

The spooler process receives an end of file on $RECEIVE only when all processes that opened the spooler process have closed it.

$RECEIVE as an Input/Output File

In this case, the server opens $RECEIVE in the I-O mode to receive requests and reply to them through the same file. Each request is acted upon and paired with a message sent back to the requester in response to the request message:



VST1406.vsd

A requester might contain the following statements:

```
    ...
CHARACTER*80 request
CHARACTER*132 reply
    ...
OPEN (UNIT=1,FILE='$SERVE',SPACECONTROL='NO')
    ...
READ (UNIT=1,PROMPT=request) reply
    ...
```

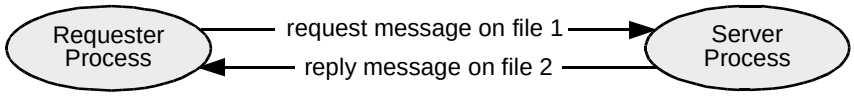
Note that a READ from a process with a PROMPT causes a WRITEREAD to be executed. The SPACECONTROL='NO' specifier in the OPEN statement ensures that all the characters transmitted are treated as data, and prevents extraneous message traffic.

The server receives request messages from $RECEIVE, performs the requested tasks, and replies to the message. The following code shows an example of a server:

```
          ...
?RECEIVE (OPEN 2, MAXREPLY 132)
          ...
   CHARACTER*80 request
   CHARACTER*132 reply
          ...
   OPEN (UNIT=1,FILE='$RECEIVE')
          ...
10 READ (UNIT=1,END=100) request
   IF (request(1:3) .EQ. 'ADD') THEN
          ...          <-- Code to process
          ...          <-- request and
          ...          <-- construct reply
   END IF
          ...
   WRITE (UNIT=1) reply
   GO TO 10
          ...
100 STOP
   END
```

$RECEIVE as Separate Input/Output Files

A server can connect $RECEIVE to one unit for input and to more than one unit for output.



VST1407.vsd

The requester is the same as in the prior example. The server code is also the same, except that there is another OPEN statement and the responding WRITE statement is changed:

```
?RECEIVE (OPEN 2, MAXREPLY 132)
         ...
   CHARACTER*80 request
   CHARACTER*132 reply
         ...
   OPEN (UNIT=1,FILE='$RECEIVE',MODE='INPUT')
   OPEN (UNIT=2,FILE='$RECEIVE',MODE='OUTPUT')
         ...
10 READ (UNIT=1,END=100) request
   IF (request(1:3) .EQ. 'ADD') THEN
         ...         <-- Code to
         ...         <-- process request
   END IF
         ...
   WRITE (UNIT=2) reply
   GO TO 10
         ...
100 STOP
   END
```

During program testing, you might use this approach and assign the input and output units to separate disk files to test program logic without the complications of asynchronous processing. When testing is completed, you can reassign both the input unit and all the output units to $RECEIVE.

READ Statement With $RECEIVE

The control list of the READ statement includes a SOURCE specifier that enables a server to determine the identity of, and information about, a requester.

The source specifier has the form:

```
SOURCE = iarr
```

where *iarr* is an integer array of at least 16 elements.

The layout of *iarr* if you specify ENV COMMON differs from the layout of iarr if you specify ENV OLD or you do not specify an ENV directive. Table 14-1 on page 14-10

shows the contents of iarr after a read operation that specifies SOURCE (declared as SOURCE(16) for this example) completes successfully.

Table 14-1. Layout of Request Message From $RECEIVE Returned on READ Statement

ENV OLD	ENV COMMON	Meaning
SOURCE(1)	SOURCE(1)	System flag: -1 if system message, 0 if user message
SOURCE(2)	SOURCE(2)	Entry number of your process in the Guardian requester table
SOURCE(3)	SOURCE(3)	Message number of your process in the Guardian requester table
SOURCE(4)	SOURCE(4)	Opener's file number for this process
SOURCE(5:8)	SOURCE(5:14)	If ENV OLD is specified, contains the opener's process identification (CRTPID) (character string) If ENV COMMON is specified, contains the opener’s process identification (process handle)
SOURCE(9)	SOURCE(15)	Number of bytes the requester expects to receive in the reply to this message
SOURCE(10:16)	SOURCE(16)	Reserved for future use

Using the READ Statement PROMPT Specifier

If the unit specified in a READ statement is a terminal or process, and the READ statement includes a PROMPT specifier, FORTRAN calls the WRITEREAD system procedure, rather than the READ system procedure. This enables a requester to send a message (the prompt) to a terminal or another process and wait for a reply.

For a full description of the WRITEREAD system procedure, see the *Guardian Procedure Calls Reference Manual*.

WRITE Statement With $RECEIVE

The WRITE statement can include either or both of the following specifiers for use with $RECEIVE:

- MSGNUM = *msg-exp*

  msg-exp is the value from SOURCE(3) of the SOURCE array returned to your program when it read the message for which the current WRITE statement is the reply. The requester process uses *msg-exp* to match your program’s reply with the original message sent by the requester. This is particular important to a requester if it has written more than one message to your process.

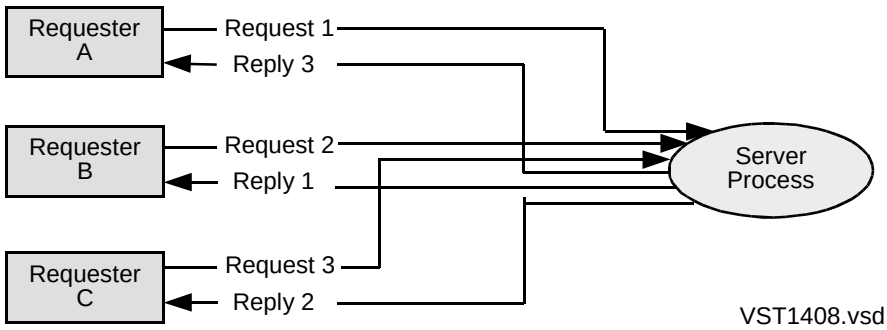
- REPLY = *rep-exp*

Your server process returns a Guardian file-system error code in *rep-exp*, or zero if no error occurred. The value you store for rep-exp is the value returned to the requester if it calls the FILEINFO or FILE_GETINFO_ system procedure.

Message Queuing

The preceding examples of the uses of $RECEIVE show server processes that respond to each request as it is received. An advanced method of interprocess communication involves message queuing. Figure 14-5 illustrates a server that can accumulate requests and respond to them in a sequence of its own choosing, which may or may not be the same as that in which they were received.

Figure 14-5. A Queued Server



In the following example, the server distributes data records from a file to different requesters. Each requester states in its request, via the PROMPT specifier of a READ statement, the keys of the records in which it is interested. The server can accommodate up to four requesters; each requester can ask for up to four distinct record keys. Both of these limits are fixed by a PARAMETER statement in the server. Example 14-1 on page 14-12 shows the two requesters. Example 14-2 on page 14-17 shows the server.

Example 14-1. Example Requesters R1 and R2 for Queued Server

```
1     ?PAGE "REQUESTER NUMBER 1 FOR QUEUED SERVER"
2     ?LOGICAL*2
3           PROGRAM requester1
4           IMPLICIT INTEGER*2 (a-z)
5           CHARACTER*8 prompt
6           DIMENSION records(4)
7           DIMENSION rec(2)
8           DATA records /1,3,5,7/
9
10         WRITE (prompt,11) records
11      11 FORMAT (4A2)
12         OPEN (UNIT=5,SPACECONTROL='NO')
13         OPEN (UNIT=6)
14         WRITE (6,22)
15      22 FORMAT ('1R1 ABOUT TO MAKE FIRST REQUEST.')
16
17      10 CONTINUE
18         READ (5,END=100,PROMPT=prompt) rec
```

Example 14-1. Example Requesters R1 and R2 for Queued Server

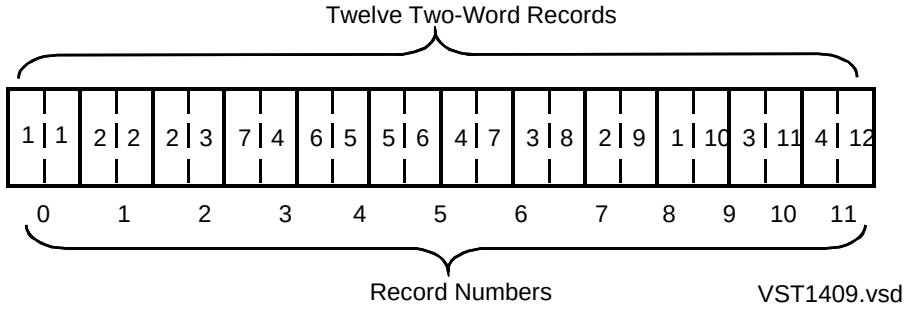
```
19          WRITE (6,33) records, rec
20       33 FORMAT (' R1',4I5, ' *',2I5)
21          GO TO 10
22
23      100 CONTINUE
24          CLOSE (UNIT=5)
25          CLOSE (UNIT=6)
26          STOP 'R1 STOPPING'
27          END

1     ?PAGE "REQUESTER NUMBER 2 FOR QUEUED SERVER"
2     ?LOGICAL*2
3           PROGRAM requester2
4           IMPLICIT INTEGER*2 (a-z)
5           CHARACTER*6 prompt
6           DIMENSION records(3)
7           DIMENSION rec(2)
8           DATA records /2,4,6/
9
10          WRITE (prompt,11) records
11       11 FORMAT (3A2)
12          OPEN (UNIT=5,SPACECONTROL='NO')
13          OPEN (UNIT=6)
14          WRITE (6,22)
15       22 FORMAT ('1R2 ABOUT TO MAKE FIRST REQUEST.')
16
17       10 CONTINUE
18          READ (5,END=100,PROMPT=prompt) rec
19          WRITE (6,33) records, rec
20          33 FORMAT (' R2',3I5, ' *',2I5)
21          GO TO 10
22
23      100 CONTINUE
24          CLOSE (UNIT=5)
25          CLOSE (UNIT=6)
26          STOP 'R2 STOPPING'
27          END 30
```

The server opens a disk file of two-word records (the first word is the key; the second, the associated data) and reads sequentially from it. If a requester has expressed interest in the current record, the server sends it to that requester. Since the order in which requests arrive is unpredictable, the server must save requests for records that have not yet been read, until those records become available.

The server, after initialization, repeatedly reads from its data file. For each record, it determines if a request for that record has been received already. If one has, the server updates its list of pending requests and sends that record to the appropriate requester.

If not, it must read requests until one for the current record arrives; it stores other requests in its list of pending requests for future use. When the data file is exhausted, the server returns an end of file for each stored request, and continues to return end of file to all new requests. As each requester receives end of file, it closes the server and terminates with a message to the home terminal. When the last requester has closed the server, the server also terminates.

Assuming that the object program for the server is named $DIST, that there are two requesters, $R1 and $R2, and that requesters read the following records,



a sample execution of this requester/server application might appear on the home terminal screen as follows:

```
1> RUN QSO /NAME $DIST, CPU 0, IN $RECEIVE, OUT $RECEIVE/
2> RUN R1O /NAME $R1, CPU 0, IN $DIST/
3> RUN R2O /NAME $R2, CPU 0, IN $DIST/
R1 ABOUT TO MAKE FIRST REQUEST.
4> PAUSE
R2 ABOUT TO MAKE FIRST REQUEST.
R1   1   3   5    7 *   1    1
R2   2   4   6      *   2    2
R2   2   4   6      *   2    3
R1   1   3   5    7 *   7    4
R2   2   4   6      *   6    5
R1   1   3   5    7 *   5    6
R2   2   4   6      *   4    7
R1   1   3   5    7 *   3    8
R2   2   4   6      *   2    9
R1   1   3   5    7 *   1   10
R2   2   4   6      *   4   12
R1   1   3   5    7 *   3   11
R2 STOPPING
R1 STOPPING

END OF QUEUEING SERVER RUN
5>
```

The source for the queued server is shown in Example 14-2 on page 14-17.

The UNIT directive at line 2 assigns unit 7 to the data file. When you run the server, you must specify $RECEIVE for both the IN file and the OUT file.

The RECEIVE directive at line 3 defines the saved reply table as follows:

- OPEN 4: Not more than four processes are allowed to have the server process open at any given time. Any additional OPEN attempts will be refused with file management error 12.
- QDEPTH 4: The server may read up to four requests without responding to any. An attempt to read another request while four requests are outstanding results in a file management error 74.
- SYNCDEPTH 2: The FORTRAN run-time library routines save two replies per requester to support NonStop requester processes. An attempt by a requester to open the server using a SYNCDEPTH greater than two will be refused with a file management error 28.
- MAXREPLY 200: The longest reply the server can issue is 200 characters.
- SYSMSG: The server will process system messages itself, rather than having the run-time support system reply to them automatically.

At lines 32 through 34, the server initializes its list of pending requests. At lines 35 through 37, it opens its files.

Statement label 10 at line 38 is the beginning of the server's main loop. At line 39, it reads the next record from its data file. The MATCHOLD function, invoked at line 41, determines whether a request for the current record has already been received. If so, then:

1. The server updates its list of pending requests to show that the matching request has been satisfied.
2. The server writes the record to $RECEIVE (line 45). The MSGNUM specifier identifies the request for which this is the reply; it was obtained when the request was read (line 49).
3. Processing returns to the beginning of the main loop.

Example 14-2. Example Queued Server (Part 1 of 3)

```
1     ?PAGE "QUEUED SERVER -- DISTRIBUTOR"
2     ?UNIT (7, QSD)
3     ?RECEIVE (OPEN 4,QDEPTH 4,SYNCDEPTH 2,MAXREPLY 200,SYSMSG)
4           PROGRAM qserver
5           IMPLICIT INTEGER*2 (a-z)
6           LOGICAL matchnew
7           PARAMETER (norqr=4, destrqr=4)
8           DIMENSION messno(norqr), request(norqr), record(66)
9           DIMENSION dest(norqr,destrqr), dest this rqr(norqr)
10          COMMON /block/messno, request, record, dest, dest this rqr
11          DIMENSION source(16)
12          LOGICAL system
13    C -- Compile-time TOGGLE 1 is FALSE if ENV OLD, TRUE if ENV COMMON
14    ?IFNOT 1
15          INTEGER rqr, msgnum, fileno, procname(4), readcount
16          EQUIVALENCE (source(1), system),
17         1            (source(2), rqr),
18         2            (source(3), msgnum),
19         3            (source(4), fileno),
20         4            (source(5), procname),
21         5            (source(9), readcount)
22    ?ENDIF 1
23    ?IF 1
24          INTEGER rqr, msgnum, fileno, procname(10), readcount
25          EQUIVALENCE (source(1), system),
26         1            (source(2), rqr),
27         2 (source(3), msgnum),
28         3 (source(4), fileno),
29         4 (source(5), procname),
30         5 (source(15), readcount)
31    ?ENDIF 1
32          CALL set array (messno, norqr, -1)
33          CALL set array (dest this rqr, norqr, 0)
34          CALL set array (dest, norqr*destrqr, -1)
35          OPEN (UNIT=5)
36          OPEN (UNIT=6)
37          OPEN (UNIT=7)
```

Example 14-2. Example Queued Server (Part 1 of 3)

```
38       10 CONTINUE
39          READ (UNIT=7,FMT=11,END=60) record(1), record(2)
40       11 FORMAT (2I5)
41          rqr = matchold (record, dest, dest this rqr, no rqr)
42          IF (rqr .NE. 0) THEN
43              messno(rqr) = -1
44              dest this rqr(rqr) = 0
45              WRITE (UNIT=6, msgnum=messno(rqr)) record
46              GO TO 10
47          END IF
```

If a request for the current record has not been received, the server must continue to read incoming requests until it finds one. This is done in an inner loop starting at statement label 20 (line 48).

At line 49, the server reads a message from $RECEIVE. The LENGTH specifier obtains the number of bytes actually read; the SOURCE specifier obtains information, in the 16-word array SOURCE, that identifies the requester. Lines 15 through 21 specify the layout of the SOURCE array for programs compiled with ENV OLD in effect or programs that do not specify an ENV directive. Lines 24 through 30 specify the layout of the SOURCE array for programs compiled with ENV COMMON in effect. If you specify ENV OLD for your program or you do not specify an ENV directive, you do not need to specify toggle 1. FORTRAN compile-time toggles are FALSE by default. If you specify ENV COMMON for a program unit, you must set compile-time toggle 1 with a SETTOG directive in order to have the correct layout of the SOURCE array.

The server handles system messages read from $RECEIVE in lines 51 through 58. If the server receives an OPEN message and the opener has specified a nowait depth greater than 1, the server replies with file system error code 28; otherwise, it replies with zero. The WRITE statement at line 56 sends a null reply to the message identified by the MSGNUM specifier; the sender receives the value of the REPLY specifier as the file system error code for the operation. Control then returns to the beginning of the inner loop to read another message from $RECEIVE.

If the current message is not a system message, the server determines if it is a request for the current data record. The MATCHNEW function invoked at line 60 accomplishes this. If the message is a request for the current record, the server sends the record to the requester. It then returns control to the beginning of the main loop to read another data record.

If the current message is a request, but not for the current record, the server must save the request in its list of pending requests until the specified record is read; this is accomplished at lines 64 through 69. The server then returns control to the beginning of the inner loop to read another incoming message.

When the end of the data file is reached, control passes to statement label 60 (line 72). Here, in a loop from line 73 to line 77, the server sends an end-of-file indication (error code 1) to each requester that has an outstanding request.

Example 14-3. Example Queued Server (Part 2 of 3)

```
48      20  CONTINUE
49          READ (UNIT=5, SOURCE=source, LENGTH=length) request
50          IF (system) THEN
51             IF (request(1) .EQ. -30 .AND.
52                + MOD(request(2),16) .GT. 1) THEN
53                replycod = 28
54             ELSE
55                replycod = 0
56             END IF
57             WRITE (UNIT=6, MSGNUM=msgnum, REPLY=replycod)
58             GO TO 20
59          END IF
60          IF (matchnew(request, length/2, record)) THEN
61             WRITE (UNIT=6, MSGNUM=msgnum) record
62             GO TO 10
63          ELSE
64             messno(rqr) = msgnum
65             limit = length/2
66             dest this rqr(rqr) = limit
67             DO 30 I = 1, limit
68                dest(rqr,I) = request(I)
69      30  CONTINUE
70          END IF
71          GO TO 20
72      60  CONTINUE
73             DO 70 I = 1,norqr
74                IF (messno(I) .GE. 0) THEN
75                   WRITE (UNIT=6,REPLY=1, MSGNUM=messno(I))
76                END IF
77      70  CONTINUE
78      90  CONTINUE
79          READ (UNIT=5,END=100, SOURCE=source) request
80          IF (system) THEN
```

Example 14-3. Example Queued Server (Part 2 of 3)

```
81              IF (request(1) .EQ. -30 .AND.
82         +       MOD(request(2),16) .GT. 1) THEN
83                 replycod = 28
84              ELSE
85                 replycod = 0
86              END IF
87           ELSE
88                replycod = 1
89           END IF
90          WRITE (UNIT=6, MSGNUM=msgnum, REPLY=replycod)
91          GO TO 90
92      100 CONTINUE
93          CLOSE (UNIT=5)
94          CLOSE (UNIT=6)
95          CLOSE (UNIT=7)
96          STOP ' END OF QUEUEING SERVER RUN'
97          END
```

Finally, control passes to the loop beginning at statement label 90 (lines 78 through 91), which reads and processes messages received via $RECEIVE. This loop treats system messages as before (lines 51-58), but sends an end-of-file indication to all user requests.

When the last opener has closed the server, the READ statement at line 79 receives an end-of-file indication; control then passes to label 100 (line 92). After closing all of its files, the server terminates its run with a stop message.

Example 14-4. Example Queued Server (Part 3 of 3)

```
98      ?PAGE          "MATCHOLD FUNCTION"
99           FUNCTION matchold (record, dest, dest this rqr,no rqr)
100          INTEGER record(1), norqr, rqr
101          INTEGER dest(norqr,1), dest this rqr(1)
102          matchold = 0
103          DO 20 rqr = 1, no rqr
104             LIMIT = dest this rqr(rqr)
105             DO 10 I = 1, LIMIT
106               IF (record(1) .EQ. dest(rqr,I)) THEN
107                  matchold = rqr
```

Example 14-4. Example Queued Server (Part 3 of 3)

```
108              END IF
109         10 CONTINUE
110       20 CONTINUE
111          RETURN
112          END
113     ?PAGE      "MATCHNEW FUNCTION"
114          LOGICAL FUNCTION matchnew(request, request words, record)
115          INTEGER request(1), request words, record(1)
116          LOGICAL answer
117          answer = .FALSE.
118          DO 10 I = 1, request words
119             IF (request(I) .EQ. record(1)) THEN
120                answer = .TRUE.
121             END IF
122       10 CONTINUE
123          matchnew = answer
124          RETURN
125          END
126     ?PAGE     "SET ARRAY SUBROUTINE"
127          SUBROUTINE set array (array, array elements, value)
128          INTEGER array(1), array elements, value
129          DO 10 I = 1, array elements
130          array(I) = value
131       10 CONTINUE
132          RETURN
133          END
```

15 Utility Routines

The FORTRAN run-time library contains utility routines supplied by HP. These routines enable FORTRAN applications to:

- Invoke system routines that are not available through standard FORTRAN constructs.
- Fetch, alter, or delete the contents of PARAM, ASSIGN, and startup messages.

The latter routines comprise the Saved Message Utility (SMU).

Topics covered in this section include:

System-Related Routines

Table 15-1 lists routines that you can use to access various system-level functions.

Each routine is designed to be used either in modules in which you specify ENV OLD or in modules in which you specify ENV COMMON but never both. For information about the ENV directive, see Section 10, Compiler Directives.

Table 15-1. FORTRAN Run-Time Utility Routines (page 1 of 2)

Routine	ENV	Action
FORTRANCOMPLETION	OLD	Enables a FORTRAN program to specify completion codes and related information when it terminates.
FORTRAN_COMPLETION_	COMMON	Enables a FORTRAN program to specify completion codes and related information when it terminates.
FORTRAN_CONTROL_	COMMON	Calls the CONTROL system procedure unless buffered spooling has been successfully initiated for the file, in which case FORTRAN_CONTROL_ calls the SPOOLCONTROL spooler procedure.

Table 15-1. FORTRAN Run-Time Utility Routines (page 2 of 2)

Routine	ENV	Action
FORTRAN_SETMODE_	COMMON	Calls the SETMODE system procedure unless buffered spooling has been successfully initiated for the file, in which case FORTRAN_SETMODE_ calls the SPOOLSETMODE spooler procedure.
FORTRAN_SPOOL_OPEN_	COMMON	Provides level-1, level-2, or level-3 access to the HP spooler. FORTRAN_SPOOL_OPEN_ combines the functionality of a FORTRAN OPEN statement and a subsequent FORTRANSPOOLSTART routine.
FORTRANSPOOLSTART	OLD	Provides level-2 and level-3 access to the HP spooler. Your program must execute an OPEN statement to open the spooler file before it calls FORTRANSPOOLSTART.
SSWTCH	COMMON	Returns the value of a specified program switch.

FORTRANCOMPLETION Routine

FORTRANCOMPLETION allows a FORTRAN program to specify completion codes and related information when it terminates.

FORTRANCOMPLETION performs the same activities (closing files, displaying a message on the home terminal, and so forth) as the FORTRAN STOP statement. Use FORTRANCOMPLETION instead of a FORTRAN STOP statement if you need to specify one or more FORTRANCOMPLETION arguments to the STOP or ABEND procedures.

```
CALL FORTRANCOMPLETION [ ( [ abend-or-stop ]
                         [, [ message-length ]
                         [, [ message ]
                         [, [ completion-code ]
                         [, [ termination-info ]
                         [, [ spi-ssid ]
                         [, [ text-length ]
                         [, [ text ] ]]] ]]] ] ) ]
```

a*bend-or-stop*

is an integer expression that specifies whether to call the STOP or ABEND system procedure to terminate execution. If *abend-or-stop is zero*, or if you omit *abend-or-stop*, FORTRANCOMPLETION calls STOP. Otherwise, it calls ABEND.

message-length

is an integer expression that specifies the length of *message*. If you omit *message-length*, the actual length of *message* is used.

message

is a character expression that the FORTRAN run-time library displays on your process's home terminal. *message* must be at least *message-length* characters if you specify *message-length*. If you do not specify *message-length*, the actual length of *message* is displayed. FORTRANCOMPLETION displays a maximum of 80 characters.

completion-code

is an integer expression whose value is passed as the *completion-code* parameter to STOP or ABEND.

termination-info

is an integer expression whose value is passed as the *termination-info* parameter to STOP or ABEND.

spi-ssid

is a character expression whose length is at least 12 characters and whose value is passed as the *spi-ssid* parameter to STOP or ABEND. *spi-ssid* is a subsystem ID (SSID) that identifies the subsystem that defines termination-info. For further information about subsystem IDs, see the *SPI Programming Manual*.

text-length

is an integer expression whose value is the number of characters in text. If you omit this argument, the actual length of *text* is used.

text

is a character expression whose length is at least *text-length* characters and whose value is passed as the text parameter to STOP or ABEND. If you do not specify *text-length*, FORTRANCOMPLETION uses the actual length of *text*.

Considerations

- Use FORTRANCOMPLETION only in programs that specify an ENV OLD directive. If you call FORTRANCOMPLETION in a program that specifies ENV COMMON, the FORTRAN compiler does not report a warning or an error, but the run-time library reports an error and terminates your program.
- All the arguments are optional. You can use or omit them in any combination.

- Executing a FORTRAN STOP statement is equivalent to calling FORTRANCOMPLETION, with all its arguments omitted (except possibly *message*). For example:

  STOP is equivalent to `CALL FORTRANCOMPLETION`

  STOP *msg* is equivalent to `CALL FORTRANCOMPLETION(, , msg)`

- Include the *spi-ssid* parameter if you use *termination-info* or text. *spi-ssid* must be a RECORD data structure or a type CHARACTER*12 variable or expression. (The corresponding parameter of STOP and ABEND must be a six-element type INTEGER*2 array.)

- If *spi-ssid* is a RECORD data structure, and if the subsystem being identified is, for example, the C10 version of NonStop subsystem number 123, *spi-ssid* must be declared and defined as follows:

```
RECORD subsystem
   CHARACTER * 8 organization
   INTEGER * 2 number
   CHARACTER * 2 version
   END RECORD


subsystem^organization = 'TANDEM'
subsystem^number = 123
subsystem^version (1: 1) = 'C'
subsystem^version (2: 2) = CHAR (10)
```

- If *spi-ssid* is a CHARACTER variable, it must be declared and defined for the previous example as follows:

```
CHARACTER * 12 ssid


ssid ( 1: 8) = 'TANDEM'
ssid ( 9: 10) = CHAR (0) // CHAR (123)
ssid (11: 12) = 'C' // CHAR (10)
```

  The variable names used here are only examples. Your programs can use any names you want, provided the layout of the data conforms to the record description provided here.

- FORTRANCOMPLETION must be named in a GUARDIAN directive in every compilation that refers to it.

- FORTRANCOMPLETION performs the same activities (closing files, displaying *message* on the home terminal, and so forth) as the FORTRAN STOP statement.

Then it calls the STOP or ABEND system procedure, as indicated by *abend-or-stop*, passing its *completion-code* through *text* arguments to STOP or ABEND.

- The *message* argument corresponds to the *message* option in the FORTRAN STOP statement. If you specify message, FORTRANCOMPLETION displays *message* on the home terminal. The *text* argument is intended for a different use. FORTRANCOMPLETION passes *text* to STOP or ABEND, which in turn stores it into the STOP or ABEND system message that the operating system sends to the ancestor process of the terminating process.
- You must ensure that the combination of parameters and their values meet the expectations of the STOP or ABEND procedure. Neither the FORTRAN compiler nor the run-time library validate the arguments. For information about the STOP and ABEND system procedures, see the *Guardian Procedure Calls Reference Manual*.

FORTRAN_COMPLETION_ Routine

FORTRAN_COMPLETION_ allows a FORTRAN program to specify completion codes and related information when it terminates.

FORTRAN_COMPLETION_ performs the same activities (closing files, displaying a message on the home terminal, and so forth) as the FORTRAN STOP statement. Use FORTRAN_COMPLETION_ instead of a FORTRAN STOP statement if you need to specify one or more FORTRAN_COMPLETION_ arguments to the PROCESS_STOP_ procedure.

```
CALL FORTRAN_COMPLETION_ [ ( [ abend-or-stop ]
                           [, [ message-length ]
                           [, [ message ]
                           [, [ completion-code ]
                           [, [ termination-info ]
                           [, [ spi-ssid ]
                           [, [ text-length ]
                           [, [ text ] ]]] ]]] ] ) ]
```

abend-or-stop

is an integer expression that specifies whether to call PROCESS_STOP_ with the ABEND option or the STOP option. If *abend-or-stop* is zero, or if you omit *abend-or-stop*, the run-time system calls PROCESS_STOP_ with the STOP option. Otherwise, it calls PROCESS_STOP_ with the ABEND option.

message-length

is an integer expression whose value is the length of the value of *message*. If you omit this argument, FORTRAN_COMPLETION_ uses the actual length of *message*.

message

is a character expression that the FORTRAN run-time library writes to the standard log file. *message* must be at least message-length characters if you specify *message-length*. If you do not specify *message-length*, the actual length of *message* is displayed. FORTRAN_COMPLETION_ displays a maximum of 80 characters.

completion-code

is an integer expression whose value is passed as the *completion-code* parameter to PROCESS_STOP_.

termination-info

is an integer expression whose value is passed as the *termination-info* parameter to PROCESS_STOP_.

spi-ssid

is a character expression whose length is at least 12 characters and whose value is passed as the *spi-ssid* parameter to PROCESS_STOP_. *spi-ssid* is a subsystem ID (SSID) that identifies the subsystem that defines termination-info. For further information about subsystem IDs, see the *SPI Programming Manual*.

text-length

is an integer expression whose value is the length of the value of *text*. If you omit this argument, the actual length of *text* is used.

text

is a character expression whose length is at least *text-length* characters and whose value is passed as the *text* parameter to PROCESS_STOP_. If you do not specify *text-length*, FORTRAN_COMPLETION_ uses the actual length of *text*.

Considerations

- Use FORTRAN_COMPLETION_ only in programs that specify an ENV
- COMMON directive. If you call FORTRAN_COMPLETION_ in a program that specifies ENV OLD, the FORTRAN compiler does not report a warning or an error, but the run-time library reports an error and terminates your program.
- All the arguments are optional. You can use or omit them in any combination.
- Executing a FORTRAN STOP statement is equivalent to calling FORTRAN_COMPLETION_, with all its arguments omitted (except possibly message). For example:

  `STOP` is equivalent to `CALL FORTRAN_COMPLETION_`

STOP msg is equivalent to CALL FORTRAN_COMPLETION_(, , *msg*)

- Include *spi-ssid* if you use *termination-info* or text. *spi-ssid* must be a RECORD data structure or a type CHARACTER*12 variable or expression. (The corresponding parameter of PROCESS_STOP_ must be a six-element type INTEGER*2 array.)
- If *spi-ssid* is a RECORD data structure, and if the subsystem being identified is, for example, the C10 version of NonStop subsystem number 123, *spi-ssid* must be declared and defined as follows:

```
RECORD subsystem
   CHARACTER * 8 organization
   INTEGER * 2 number
   CHARACTER * 2 version
   END RECORD


subsystem^organization = 'TANDEM '
subsystem^number = 123
subsystem^version (1: 1) = 'C'
subsystem^version (2: 2) = CHAR (10)
```

- If *spi-ssid* is a CHARACTER variable, it must be declared and defined for the previous example as follows:

```
CHARACTER * 12 ssid


ssid ( 1: 8) = 'TANDEM '
ssid ( 9: 10) = CHAR (0) // CHAR (123)
ssid (11: 12) = 'C' // CHAR (10)
```

  The variable names used here are only examples. Your programs can use any names you want, provided the layout of the data conforms to the record description provided here.

- FORTRAN_COMPLETION_ must be named in a GUARDIAN directive in every compilation that refers to it.
- FORTRAN_COMPLETION_ performs the same activities (closing files, displaying *message* on the home terminal, and so forth) as the FORTRAN STOP statement. Then it calls the PROCESS_STOP_ procedure specifying the STOP or ABEND option, as indicated by *abend-or-stop* and passing its *completion-code* through *text* arguments to PROCESS_STOP_.
- The *message* argument corresponds to the *message* option in the FORTRAN STOP statement. If you specify *message*, FORTRAN_COMPLETION_ writes

message to the standard log file. The *text* argument is intended for a different use. FORTRAN_COMPLETION_ passes text PROCESS_STOP_, which in turn stores it into the STOP or ABEND system message that is sent to the ancestor process of the terminating process.

- You must ensure that the combination of parameters and their values meet the expectations of the PROCESS_STOP_ system procedure. Neither the FORTRAN compiler nor the run-time library validates the arguments. For information about the SPROCESS_STOP_ system procedure, see the *Guardian Procedure Calls Reference Manual*.

FORTRAN_CONTROL_ Routine

FORTRAN_CONTROL_ calls the CONTROL system procedure unless buffered spooling has been successfully initiated for the file, in which case FORTRAN_CONTROL_ calls the SPOOLCONTROL spooler procedure.

FORTRAN_CONTROL_ enables you to issue control operations to spooler collectors with a minimum of programming effort.

```
CALL FORTRAN_CONTROL_ ( unit-number
                      , [ error-return ]
                      , operation
                     [, [ param ] ] )
```

unit-number

is an INTEGER*2 expression whose value is the FORTRAN unit number to which to send the CONTROL operation.

error-return

is an INTEGER*2 variable in which FORTRAN_CONTROL_ returns an error code.

If *error-return* is zero, FORTRAN_CONTROL_ successfully issued a CONTROL or SPOOLCONTROL operation.

If *error-return* is less than 10000, its value is a file system error code. For information about file system errors, see the *Guardian Procedure Errors and Messages Manual* . For information about errors returned by the SPOOLCONTROL spooler procedure, see the *Spooler Programmer's Guide*.

If *error-return* is greater than 10000, its value is a FORTRAN run-time error code to which the FORTRAN run-time library has added 10000. To determine the actual error code, subtract 10000 from the value returned in *error-return* and see Appendix G, Run-Time Diagnostic Messages.

For example, if *error-return* is 250, the error is a file system error—the file referenced by *unit-number* is on a node that is no longer accessible. If *error-return* is 10064, you must subtract 10000 from *error-return* to produce error 64: the unit-number that you specified is not associated with a currently open file.

operation

is the code to send to the device referenced by *unit-number*. For specific operation codes, see the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

param

specifies the value of the parameter to the CONTROL operation you specify. For specific *param* values for each CONTROL operation, see the CONTROL procedure in the *Guardian Procedure Calls Reference Manual*.

FORTRAN_CONTROL_ calls the CONTROL system procedure unless unit-number is a spooler collector doing level-3 spooling, in which case FORTRAN_CONTROL_ calls the SPOOLCONTROL spooler procedure.

Considerations

- If an error occurs and you do not specify error-return, the FORTRAN run-time library terminates your program.
- Use FORTRAN_CONTROL_ only in programs that specify an ENV COMMON directive. The FORTRAN compiler does not report an error if you use FORTRAN_CONTROL_ with ENV OLD, but the FORTRAN run-time library reports an error if your program executes FORTRAN_CONTROL_ in a module that specifies ENV OLD.

FORTRAN_SETMODE_ Routine

FORTRAN_SETMODE_ calls the SETMODE system procedure unless buffered spooling has been successfully initiated for the file, in which case FORTRAN_SETMODE_ calls the SPOOLSETMODE spooler procedure. FORTRAN_SETMODE_ enables you to issue setmode operations to spooler collectors with a minimum of programming effort.

```
CALL FORTRAN_SETMODE_ ( unit-number
                      , [ error-return ]
                      , function
                     [, [ param1 ]
                     [, [ param2 ] ]] )
```

unit-number

is an INTEGER*2 expression whose value is the FORTRAN unit number to which to send the SETMODE command.

error-return

is an INTEGER*2 variable in which FORTRAN_SETMODE_ returns an error code.

If *error-return* is zero, FORTRAN_SETMODE_ successfully issued a SETMODE or SPOOLSETMODE operation.

If *error-return* is less than 10000, its value is a file system error code. For information about file system errors, see the *Guardian Procedure Errors and Messages Manual* . For information about errors returned by the SPOOLSETMODE spooler procedure, see the *Spooler Programmer's Guide*.

If *error-return* is greater than 10000, its value is a FORTRAN run-time error code to which the FORTRAN run-time library has added 10000. To determine the actual error code, subtract 10000 from the value returned in *error-return* and see Appendix G, Run-Time Diagnostic Messages.

For example, if *error-return* is 250, the error is a file system error—the file referenced by *unit-number* is on a node that is no longer accessible. If *error-return* is 10064, you must subtract 10000 from *error-return* to produce error 64: the *unit-number* that you specified is not associated with a currently open file.

function

specifies the SETMODE function to execute. For specific commands, see the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

param1

specifies the value of the first parameter to the specific SETMODE function you are executing. For specific *param* values for each SETMODE function, see the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

param2

specifies the value of the second parameter to the specific SETMODE function you are executing. For specific *param2* values for each SETMODE function, see the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

FORTRAN_SETMODE_ calls the SETMODE system procedure unless *unit-number* is a spooler collector doing level-3 spooling in which case FORTRAN_SETMODE_ calls the SPOOLSETMODE spooler procedure.

Considerations

Use FORTRAN_SETMODE_ only in programs that specify an ENV COMMON directive. The FORTRAN compiler does not report an error if you use FORTRAN_SETMODE_ with ENV OLD but the FORTRAN run-time library reports an error if your program executes FORTRAN_SETMODE_ in a module that specifies ENV OLD.

FORTRAN_SPOOL_OPEN_ Routine

FORTRAN_SPOOL_OPEN_ provides level-1, level-2, and level-3 access to the HP spooler from a FORTRAN program. You can use FORTRAN_SPOOL_OPEN_ only in programs that specify ENV COMMON.

FORTRAN attempts to use level-3 spooling for any file that opens a spooler collector if PARAM BUFFERED-SPOOLING OFF is not in effect. Calling FORTRAN_SPOOL_OPEN_ enables you to use any spooling level and also to set specific spooler attributes for the file.

```
CALL FORTRAN_SPOOL_OPEN_ ( unit-number
                         [, [ error-return ]
                         [, [ filename ]
                         [, [ filename-size ]
                         [, [ protect ]
                         [, [ mode ]
                         [, [ stackspec ]
                         [, [ spacecontrol ]
                         [, [ spooling-level ]
                         [, [ location ]
                         [, [ form-name ]
                         [, [ report-name ]
                         [, [ number-of-copies ]
                         [, [ page-size ]
                         [, [ flags ]
                         [, [ owner ]
                         [, [ max-lines ]
                        [, [ max-pages ] ]]] ]]] ]]] ]]] ]]] ]])
```

unit-number

is an integer expression whose value is the FORTRAN unit number of the file to spool.

error-return

is an INTEGER*2 variable in which FORTRAN_SPOOL_OPEN_ returns an error code.

If *error-return* is zero, FORTRAN_SPOOL_OPEN_ successfully opened the file with the spooler attributes you specified.

If *error-return* is less than 10000, its value is a file system error code. For information about file system errors, see the *Guardian Procedure Errors and Messages Manual* . For information about errors returned by spooler procedures, see the *Spooler Programmer's Guide*.

If *error-return* is greater than 10000, its value is a FORTRAN run-time error code to which the FORTRAN run-time library has added 10000. To determine the actual error code, subtract 10000 from the value returned in *error-return* and see Appendix G, Run-Time Diagnostic Messages.

For example, if *error-return* is 201, the error is a file system error—the file referenced by *unit-number* is on a node that is no longer accessible. If *error-return* is 10250, you must subtract 10000 from *error-return* to produce error 250: the node associated with *unit-number* that you specified is not currently accessible over the HP network.

filename

specifies the name of the spooler collector or spooler job file to open for the spooling session.

filename-size

specifies how many characters at the beginning of *filename* identify the destination spooler file.

protect

is a character expression with the value 'SHARED', 'PROTECTED', or 'EXCLUSIVE' that specifies how the file is to be shared.

mode

is a character expression with the value 'INPUT', 'OUTPUT', or 'I-O' that specifies whether to open the file for read access, write access, or read and write access. The default value (in the absence of a UNIT directive or ASSIGN command specifying otherwise) is 'I-O'.

stackspec

is a character expression with the value 'YES' or 'NO'. For additional information, see Considerations on page 15-15.

If you run your program as a NonStop process, each time you execute FORTRAN_SPOOL_OPEN_, the FORTRAN run-time library checkpoints your program environment to your backup process unless *stackspec* specifies 'NO'.

spacecontrol

spacecontrol is a character expression with the value 'YES', 'NO', or 'DEVICE' and specifies whether the first character of each output record is a control character that controls vertical spacing for an output device, or is a character of data. See “Considerations” in the description of the OPEN Statement on page 7-70.

spooling-level

specifies the spooling level to use for the file. You can specify the following values:

Spooling Level	Effect
< -1	FORTRAN_SPOOL_OPEN_ returns error 10056 (error 56), invalid parameter.
-1	The file uses level-3 spooling if the FORTRAN run-time library can obtain a buffer and PARAM BUFFERED-SPOOLING OFF is not specified. Otherwise, the file uses level-1 or level-2 spooling. Specifying -1 has the same effect as omitting the *spooling-level* parameter.
0	The file uses level-1 or level-2 spooling. The run-time library does not allocate a buffer.
> 0	The file uses level-3—buffered—spooling. Your program terminates or returns an error if the file cannot use level-3 spooling—for example, if the runtime library cannot allocate a buffer.

location

is a character expression whose value specifies the location for the spooler job. location overrides the location components of *filename*. *location* requires 16 characters. If *location* is more than 16 characters, only the first 16 characters are passed to the spooler. If *location* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of *location* when it calls SPOOLSTART. The spooler expects a two-part location name in the following format:

```
LOCATION ( 1: 1) must be "#"
LOCATION ( 2: 8) group name
LOCATION ( 9: 16) destination name
```

The group and destination names can be any combination of letters, digits, and blanks.

form-name

is a character expression whose value specifies the form name for the spooler job. *form-name* requires 16 characters. If *form-name* is more than 16 characters, only the first 16 characters are passed to the spooler. If *form-name* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of location when it calls SPOOLSTART. The spooler accepts any combination of letters, digits, and blanks.

report-name

is a character expression whose value specifies the report name for the spooler job. *report-name* requires 16 characters. If report-name is more than 16

characters, only the first 16 characters are passed to the spooler. If *report-name* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of *report-name* when it calls SPOOLSTART. The spooler accepts any combination of letters, digits, and blanks.

number-of-copies

is an integer expression whose value specifies the number of copies to print. *number-of-copies* must be in the range 1 through 32,767. The default is 1.

page-size

is an integer expression whose value specifies the number of lines per page the HP NonStop PERUSE utility uses for its PAGE and LIST commands. *page-size* must be in the range 1 through 32,767. The default is 60 lines.

flags

is an integer expression whose value specifies certain attributes for the spooler job. The value of *flags* is the sum of the values associated with the defined options, which are:

```
HOLD 0 = off, 64 = on
HOLDAFTER 0 = off, 32 = on
PRIORITY 0, 1, 2, 3, 4, 5, 6, or 7
```

The default is 4 (HOLD=off, HOLDAFTER=off, PRIORITY=4).

owner

is a character expression whose value specifies the owner of the spooler job in the format:

```
OWNER ( 1: 8) group name
OWNER ( 9: 16) user name
```

The value of the group name is a file-security system group name, and the user name is a file-security system user name. Each name must be eight characters long, with trailing blanks inserted if a name is shorter than eight characters. The item does not include a period between the names. If you specify an invalid combination of names for *owner*, error code 102 is returned in *error-return*. If *owner* is omitted, the spooler uses the owner ID of the executing process.

max-lines

is an integer expression whose value specifies the maximum number of lines allowed for the spooler job. *max-lines* must be in the range 1 through 65,534. If *max-lines* is zero or omitted, the spooler does not impose a limit on the number of lines it writes.

max-pages

is an integer expression whose value specifies the maximum number of pages for the spooler job. *max-pages* must be in the range 1 through 65,534. If *max-pages* is zero or omitted, the spooler does not impose a limit on the number of pages it writes.

Considerations

- Use FORTRAN_SPOOL_OPEN_ only in programs that specify an ENV COMMON directive. The FORTRAN compiler does not report an error if you use FORTRAN_SPOOL_OPEN_ with ENV OLD but the FORTRAN run-time library reports an error if your program executes FORTRAN_SPOOL_OPEN_ in a module that specifies ENV OLD.
- All the arguments except *unit-number* are optional. You can use or omit them in any combination.
- The FORTRAN_SPOOL_OPEN_ routine must be named in a GUARDIAN directive in every compilation that refers to it.
- You can invoke the FORTRAN_SETMODE_ and FORTRAN_CONTROL_ utility routines if you specify buffered spooling and you need to invoke operations defined by the SPOOLSETMODE or SPOOLCONTROL system routines.
- The following rules show the values to specify for *spooling-level* if *unit-number* is 6 and the spooler collector associated with unit 6 might be opened by a routine in your process that is written in a language other than FORTRAN:
  - If the original open of the file resulted in buffered spooling, a call to FORTRAN_SPOOL_OPEN_ for the same file must specify *spooling-level* > 0 or *spooling-level* = -1.
  - If the original open of the file resulted in level-1 or level-2 spooling, a call to FORTRAN_SPOOL_OPEN_ for the same file must specify *spooling-level* = 0 or *spooling-level* = -1.
  - If the original open of the file did not establish a spooling level (run-time routines determine the spooling level) then if no routine has written records to the file, you can specify any spooling level. If any routine in your process has written at least one record to the file, then all subsequent opens should specify *spooling-level* = -1. Specifying *spooling-level* = 0 or *spooling-level* > 0 might not succeed, depending on whether the run-time routines are using level-1 or level-3 spooling.

FORTRANSPOOLSTART Routine

FORTRANSPOOLSTART provides level-2 and level-3 access to the HP spooler from a FORTRAN program.

FORTRAN automatically provides level-3 spooling for a file directed to the spooler when FORTRANSPOOLSTART is not used and PARAM SPOOLOUT 0 is not in effect.

```
CALL FORTRANSPOOLSTART ( unit-number
                        [, [ error-return ]
                        [, [ options ]
                        [, [ level-3-buffer ]
                        [, [ location ]
                        [, [ form-name ]
                        [, [ report-name ]
                        [, [ number-of-copies ]
                        [, [ page-size ]
                        [, [ flags ]
                        [, [ owner ]
                        [, [ max-lines ]
                        [, [ max-pages ] ]]] ]]] ]]] ]]] )
```

unit-number

is an integer expression whose value is the FORTRAN unit number of the file to spool. If the spool file is opened before FORTRANSPOOLSTART, your program must not write to it until it calls FORTRANSPOOLSTART. If the file is not opened before calling FORTRANSPOOLSTART, FORTRANSPOOLSTART will open it.

error-return

is an INTEGER*2 variable in which FORTRANSPOOLSTART returns zero if it successfully opens the spooler, or a file system or SPOOLSTART error code if it cannot open the spooler. For explanations of spooler errors, see the *Guardian Procedure Errors and Messages Manual* and for explanations of file system errors, see the *Spooler Programmer's Guide*.

If you do not specify *error-return* and the error code is nonzero, your program stops and an error message is sent to the home terminal.

options

is an integer expression whose value specifies which level of spooling you want. If options is 0, level-1 or level-2 spooling is used for the spool file. If options is a nonzero value, level-3 spooling is used for the spool file. The FORTRAN run-time library allocates the buffer space for level-3 spooling unless you specify *level-3-buffer*.

level-3-buffer

is an INTEGER*2 array that contains at least 512 elements. This array is the buffer in which to store spool file records before sending them to the spooler. It must be allocated statically (in a common block or named in a DATA or SAVE statement) and must not be allocated in the extended data segment.

location

is a character expression whose value specifies the location for the spooler job. *location* overrides the location components of the file name you specify in an OPEN statement. *location* requires 16 characters. If location is more than 16 characters, only the first 16 characters are passed to the spooler. If *location* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of *location* when it calls SPOOLSTART. The spooler expects a two-part location name in the following format:

```
LOCATION ( 1: 1) must be "#"
LOCATION ( 2: 8) group name
LOCATION ( 9: 16) destination name
```

The group and destination names can by any combination of letters, digits, and blanks.

form-name

is a character expression whose value specifies the form name for the spooler job. *form-name* requires 16 characters. If *form-name* is more than 16 characters, only the first 16 characters are passed to the spooler. If *form-name* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of location when it calls SPOOLSTART. The spooler accepts any combination of letters, digits, and blanks.

report-name

is a character expression whose value specifies the report name for the spooler job. *report-name* requires 16 characters. If *report-name* is more than 16 characters, only the first 16 characters are passed to the spooler. If *report-name* is less than 16 characters, the FORTRAN run-time library adds blanks on the right side of *report-name* when it calls SPOOLSTART. The spooler accepts any combination of letters, digits, and blanks.

number-of-copies

is an integer expression whose value specifies the number of copies to print. *number-of-copies* must be in the range 1 through 32,767. The default is 1.

page-size

is an integer expression whose value specifies the number of lines per page the NonStop PERUSE utility uses for its PAGE and LIST commands. *page-size* must be in the range 1 through 32,767. The default is 60 lines.

flags

is an integer expression whose value specifies certain attributes for the spooler job. *flags* is the sum of the values associated with the defined options, which are:

```
HOLD 0 = off, 64 = on
HOLDAFTER 0 = off, 32 = on
PRIORITY 0, 1, 2, 3, 4, 5, 6, or 7
```

The default is 4 (HOLD=off, HOLDAFTER=off, PRIORITY=4).

owner

is a character expression whose value specifies the owner of the spooler job in the format:

```
OWNER ( 1: 8) group name
OWNER ( 9: 16) user name
```

The value of the group name is a file-security system group name, and the user name is a file-security system user name. Each name must be eight characters long, with trailing blanks inserted if a name is shorter than eight characters. The item does not include a period between the names. If you specify an invalid combination of names for *owner*, error code 102 is returned in *error-return*. If *owner* is omitted, the spooler uses the owner ID of the executing process.

max-lines

is an integer expression whose value specifies the maximum number of lines allowed for the spooler job. *max-lines* must be in the range 1 through 65,534. If *max-lines* is zero or omitted, the spooler does not impose a limit on the number of lines it writes.

max-pages

is an integer expression whose value specifies the maximum number of pages for the spooler job. *max-pages* must be in the range 1 through 65,534. If *max-pages* is zero or omitted, the spooler does not impose a limit on the number of pages it writes.

Choosing a Spooling Level

The following list describes the three levels of spooling that you can specify.

- Level-1 spooling

  FORTRANSPOOLSTART uses level-1 spooling if *options* is zero and you omit all other parameters. If you want all spooled files to use level-1 spooling, you can specify PARAM SPOOLOUT 0 when you run your program, rather than using FORTRANSPOOLSTART at all.

  Level-1 spooling is desirable when your program contains direct calls to Guardian procedures for the file, such as CONTROL and SETMODE, that do not work with level-3 spooling.

- Level-2 spooling

  FORTRANSPOOLSTART uses level-2 spooling if options is zero, you specify at least one other parameter, and you do not specify *level-3-buffer*.

  Level-2 spooling is desirable when your program contains direct calls to Guardian procedures for the file, such as CONTROL and SETMODE, that do not work with level-3 spooling but you want to establish initial spooling parameters.

- Level-3 spooling

  FORTRANSPOOLSTART uses level-3 spooling if *options* is nonzero. Your program can allocate a buffer by specifying *level-3-buffer* or you can omit *level-3-buffer*, in which case FORTRAN allocates the buffer space at runtime.

  If you want all spooled files to use level-3 spooling, you need not call FORTRANSPOOLSTART at all. FORTRAN assumes level-3 spooling by default and automatically allocates the required buffer space at run-time. However, you must use FORTRANSPOOLSTART if you want to specify spool file attributes.

Considerations

- Use FORTRANSPOOLSTART only in programs compiled with ENV OLD. The FORTRAN compiler does not report an error if you use FORTRANSPOOLSTART with ENV COMMON but the FORTRAN run-time library reports an error if your program executes FORTRANSPOOLSTART in a module that specifies ENV COMMON.
- All FORTRANSPOOLSTART arguments are optional except *unit-number*. You can use or omit them in any combination.
- The FORTRANSPOOLSTART routine must be named in a GUARDIAN directive in every compilation that refers to it.
- If the OPEN statement in which you open *unit-number* specifies the TIMED option, your program uses level-1 spooling regardless of the value you specify for *options* when you call FORTRANSPOOLSTART.

SSWTCH Routine

SSWTCH returns the current value of a program switch.

```
CALL SSWTCH ( switch-number, result)
```

switch-number

is a type INTEGER*2 variable that specifies which switch's value to return.

result

is a type INTEGER*2 variable in which SSWTCH stores the current value of switch *switch-number*.

Considerations

- Use SSWTCH only in programs compiled with ENV COMMON. The FORTRAN compiler does not report an error if you use SSWTCH with ENV OLD but the FORTRAN run-time library returns a switch value of two if your program executes SSWTCH in a module that specifies ENV OLD. See the following table for additional meanings of a switch value of two.
- You set switch values using a TACL PARAM command.

```
PARAM SWITCH- nn { ON  }
                 { OFF }
```

- SSWTCH returns:

Switch Value	Meaning
1	Switch is ON
2	Any of the following: • Switch is OFF • *switch-number* is less than 1 or greater than 15 • The program was not compiled with ENV COMMON

Saved Message Utility

The Saved Message Utility (SMU) is a collection of routines that are extensions to the HP FORTRAN product. Table 15-2 lists the SMU routines. You use these routines to save and modify the messages sent to your process by the operating system. The *Guardian Procedure Calls Reference Manual* contains a complete description of these messages.

The first part of this subsection describes how you use the SMU routines. The second part lists all the SMU routines in alphabetical order, and includes the syntax and usage considerations for each routine.

When TACL starts a process, it sends a series of messages to the process that describe the following:

- The IN file
- The OUT file
- The default volume and subvolume
- Current ASSIGN values
- Current PARAM values
- Additional text specified with the RUN command

If you want to save these messages, you can use the SAVE directive to specify which messages to save, and then use SMU routines to retrieve them. If your program initiates other processes, you can use the SAVE directive to save the messages that describe the startup environment of the parent process, and then use SMU routines to customize the startup environment for the new processes.

Table 15-2. Saved Message Utility Routines (page 1 of 2)

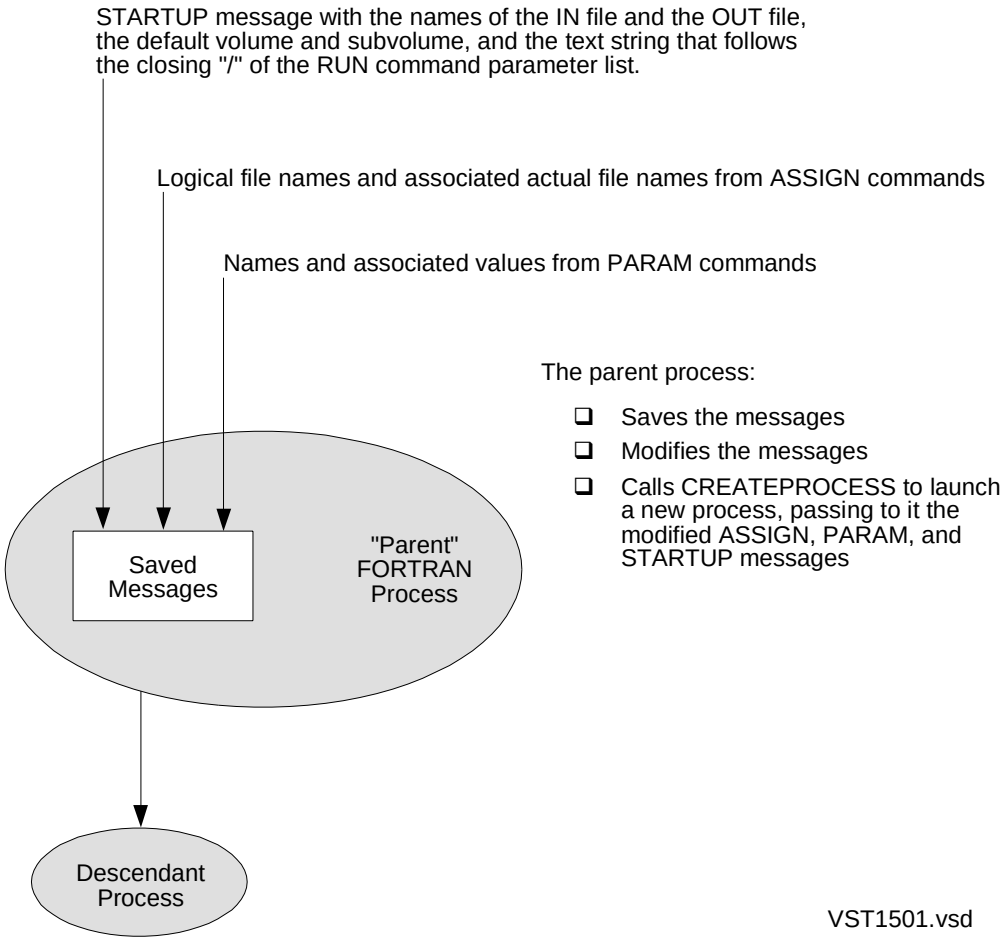
Name	Action
ALTERPARAMTEXT	Creates or replaces the value for a specific parameter name, in the PARAM message, with the option of trailing blanks in the new value.
CHECKLOGICALNAME	Checks whether an ASSIGN message with a given logical file name exists.
CHECKMESSAGE	Checks whether a specific message exists.
CREATEPROCESS	Creates a new process and sends the initial ASSIGN, PARAM, and startup messages.
DELETEASSIGN	Deletes a portion or all of an ASSIGN message.
DELETEPARAM	Deletes a portion or all the PARAM message.
DELETESTARTUP	Deletes the entire startup message.
GETASSIGNTEXT	Retrieves a portion of an ASSIGN message as text and assigns it to a string variable.

Table 15-2. Saved Message Utility Routines (page 2 of 2)

Name	Action
GETASSIGNVALUE	Retrieves a portion of an ASSIGN message as an integer and assigns it to an integer variable.
GETBACKUPCPU	Retrieves a backup CPU number from the PARAM message.
GETPARAMTEXT	Retrieves a portion of the PARAM message as text and assigns it to a string variable.
GETSTARTUPTEXT	Retrieves a portion of the startup message as text and assigns it to a string variable.
PUTASSIGNTEXT	Creates or replaces a portion of an ASSIGN message with text from a string variable.
PUTASSIGNVALUE	Creates or replaces a portion of an ASSIGN message with a value from an integer variable.
PUTPARAMTEXT	Creates or replaces a portion of a PARAM message with text from a string variable.
PUTSTARTUPTEXT	Creates or replaces a portion of the startup message with text from a string variable.

Figure 15-1 on page 15-23 illustrates how process messages are manipulated by the SMU.

Figure 15-1. Process Messages Manipulated by the SMU



Using SMU Routines

To reference an SMU routine in a program unit, you must declare it in a GUARDIAN directive. (The GUARDIAN directive is described in Section 10, Compiler Directives.) The following example shows how to declare the SMU routine GETPARAMTEXT as a Guardian procedure:

```
?GUARDIAN GETPARAMTEXT
```

You must declare the routine before the first FORTRAN statement that calls that procedure.

SMU routines operate on copies of the process creation messages that establish the execution environment for a program. Copies of these messages are not saved automatically; you must request them using the SAVE compiler directive. During process creation the messages selected by a SAVE directive are saved in an area inaccessible to the FORTRAN program.

Each saved ASSIGN message is given a unique, serially assigned message number. This number is a positive integer from one to the greatest number of saved ASSIGN messages. It is used to specifically identify each saved ASSIGN message.

SMU routines operate on specific portions of saved messages. Each portion is identified by a special string value. You must pass the string value as the portion parameter to the appropriate routine. Case is not significant in specifying string values, but you must observe the following restrictions:

- The string cannot be preceded by blanks.
- The string must be followed by a blank when the portion parameter length exceeds the string value length.
- The string value must be enclosed in single quotes.

For the content of message portions, see Saved Messages on page 15-26.

Types of SMU Routines

There are three types of SMU routines: those that obtain environment information for a process, those that change environment values, and those that delete stored environment values.

Getting Environment Information

To obtain environment information, you must use the SMU routines whose names begin with GET. Table 15-3 on page 15-25 describes these routines.

Table 15-3. SMU Routines for Obtaining Environment Information

Routine	Portion	Returns
GETASSIGNTEXT	LOGICALNAME	File description name
	TANDEMNAME	
GETASSIGNVALUE	ACCESS	Access mode
	BLKSIZE	Block size
	EXCLUSION	Exclusion code
	FILECODE	File code
	PRIEXT	Primary disk extent
	RECSIZE	Record size
	SECEXT	Secondary extent
GETPARAMTEXT	Name of the parameter whose text is to be returned	Value of parameter
GETSTARTUPTEXT	IN	Name of IN file
	OUT	Name of OUT file
	STRING	Text following run option-option list on the command line
	VOLUME	Names of default volume and subvolume

The following example uses GETASSIGNTEXT to obtain a file name:

```
?GUARDIAN GETASSIGNTEXT
      CHARACTER*63 filename
      INTEGER error
      error = GETASSIGNTEXT ('TANDEMNAME', filename, 1)
```

Changing Environment Information

The group of SMU routines that begin with PUT and the ALTERPARAMTEXT routine allow you to change environment information. The PUT routines are the exact counterparts of the GET routines. Each PUT routine enables you to insert a new value into a PARAM, ASSIGN, or startup message.

The difference between PUTPARAMTEXT and ALTERPARAMTEXT is that ALTERPARAMTEXT enables you to insert trailing blanks in the new parameter value.

The following example uses the ALTERPARAMTEXT routine to change a parameter value:

```
?GUARDIAN ALTERPARAMTEXT
      INTEGER error
      CHARACTER*20 oldparam, newparam
      READ (9,*) oldparam
      READ (*,*) newparam
      error = ALTERPARAMTEXT (oldparam, newparam, 0, 5 )
```

Deleting Environment Information

You can delete stored values and text by using the SMU routines that begin with DELETE. The following example uses the DELETESTARTUP routine to delete the entire startup message:

```
?GUARDIAN DELETESTARTUP
      INTEGER error
      error = DELETESTARTUP ( '*ALL*', 0)
```

Saved Messages

The internal data structures of the saved process-creation messages can differ slightly from the standard FORTRAN data structures. The routines operate mainly on components, or “portions,” of these saved messages. If a message component is not suitable, the routine formats it either to or from a specific external representation.

The following sections identify the different portions of saved messages.

The PARAM Message

The PARAM message contains all parameter names and the values associated with the names. The PARAM message includes:

- A count of the number of named parameters
- A list of the named parameters in the following form:
  - Length of name
  - Name
  - Length of value
  - Value

You can use SMU routines to request, modify, or delete parameter names and values. You cannot determine the number of parameters or the names of the parameters in the PARAM message.

The following SMU routines operate on the PARAM message:

```
GETPARAMTEXT        DELETEPARAM
GETBACKUPCPU        PUTPARAMTEXT
ALTERPARAMTEXT
```

The ASSIGN Messages

The ASSIGN messages contain file names and attributes that you specify using a TACL ASSIGN command. For additional information about the ASSIGN command, see Section 5, Introduction to File I/O in the HP NonStop Environment.

The following SMU routines operate upon ASSIGN messages:

```
CHECKLOGICALNAME        PUTASSIGNTEXT
CHECKMESSAGE            PUTASSIGNVALUE
GETASSIGNTEXT           DELETEASSIGN
GETASSIGNVALUE
```

Table 15-4. The Portions of the ASSIGN Message

Portion Name	Type	Identifies
LOGICALNAME	Text	Logical unit name
TANDEMNAME	Text	File name
PRIEXT	Integer	Primary extent size
SECEXT	Integer	Secondary extent size
FILECODE	Integer	File code
ACCESS	Integer	Access mode
EXCLUSION	Integer	Exclusion mode
RECSIZE	Integer	Record size
BLKSIZE	Integer	Block size

The Startup Message

The following SMU routines operate on the startup message:

```
GETSTARTUPTEXT
PUTSTARTUPTEXT
DELETESTARTUPTEXT
```

Table 15-5. The Portions of the Startup Message

Portion Name	Type	Identifies
VOLUME	Text	Default volume and subvolume names
IN	Text	Input file name
OUT	Text	Output file name
STRING	Text	The startup message's parameter string (the text that follows the RUN option list)

Checkpoint Considerations for Saved Message Utility Routines

In an HP FORTRAN program the storage space for saved messages is not directly accessible. When a NonStop process needs to change or delete any saved messages, a checkpoint list is required. The checkpoint list is a programmer-declared FORTRAN array where the saved message changes are recorded.

You must transmit the information contained in a checkpoint list to the backup of a process pair just as you would transmit the value of any critical variable by using the FORTRAN CHECKPOINT statement. The control list specifier, CPLIST, is recognized in the CHECKPOINT statement as the optional checkpoint list item.

You can provide any number of CPLIST specifiers in a single CHECKPOINT statement. The checkpoint list can include other control list items that are legal for the FORTRAN CHECKPOINT statement; for example:

```
CHECKPOINT (CPLIST = cplist1, UNIT = 3) item1, item2
```

FORTRAN interprets CPLIST1 as a checkpoint list and checkpoints the information it contains as well as the values of item1 and item2. FORTRAN adds checkpoint information to the list whenever a routine modifies a saved message. After a checkpoint, the information in CPLIST1 is "emptied". Its storage space is then available to record further message changes.

If you want changes to saved messages to be checkpointed, you must furnish a complete checkpoint list array. The list is an INTEGER*4 array. For information about how to declare CPLIST, see the CHECKPOINT Statement on page 7-15.

The required number of INTEGER*4 array elements depends on the number of operations the list must record prior to a checkpoint. This number varies depending on the routine. The description of the *cplist* parameter for each SMU routine specifies the maximum number of elements required for this array.

The first value in the DATA list must be one less than the length of the checkpoint list array. The second value must be zero. There are 101 elements in the above array. The first value is 100.

You need to supply a complete checkpoint list only if a program has a backup that must be kept current. When a record of the changes to saved messages is not required, the *cplist* parameter is 0. For example:

```
INTEGER*2 noncplist
DATA noncplist / 0 /
```

You can avoid saturating the checkpoint list and causing a routine failure by declaring large list arrays and checkpointing the information regularly.

Your program must not modify directly the contents of a checkpoint list. The SMU routines and the logic that supports the FORTRAN CHECKPOINT statement maintain checkpoint lists without the need for program action.

The checkpoint list should not be allocated memory space in such a way that it can be de-allocated between the time you save information in it and the time you send it to the backup process. That is, your program should allocate the checkpoint list statically rather than dynamically. The easiest way to do this is to put the checkpoint list array in a common block or name it in a DATA or SAVE statement. The preceding examples use DATA statements.

ALTERPARAMTEXT Routine

The ALTERPARAMTEXT routine creates or replaces a parameter value for the specified PARAM, with text from a string variable. Unlike the PUTPARAMTEXT routine, ALTERPARAMTEXT provides for parameter values with trailing spaces.

```
result = ALTERPARAMTEXT ( portion, text, cplist, size)
```

result

is an integer variable in which ALTERPARAMTEXT returns the result of the operation. See Considerations on page 15-30.

portion

is a character expression that specifies the name of the PARAM whose text is altered. The string value must be a legal PARAM name.

In the value of *portion*, the first unused character position, if any, must be a blank. Any characters including and following a blank are ignored.

text

is a character expression whose value is stored in the specified PARAM.

cplist

is a checkpoint list in which ALTERPARAMTEXT records the changes made to the message storage data space. ALTERPARAMTEXT uses a maximum of six

elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

size

is an integer expression whose value is the number of characters in text to use as the new PARAM parameter value. *size* must be a non-negative integer that is less than or equal to 255.

Considerations

ALTERPARAMTEXT returns the following values:

Return Value	Meaning
≥0	*size* characters of *text* were assigned as the new parameter value.
-1	A failure due to a logic problem. The message is unchanged. Possible logic errors are: ● The portion value is not correct. ● The size value is negative or exceeds 255. ● The total length of the new PARAM message exceeds the maximum. ● The contents of the checkpoint list parameter are inconsistent.
-2	Insufficient checkpoint list space to complete the operation. The message is unchanged.
-3	Insufficient message storage space to complete the operation. The message is unchanged.

Example

```
?GUARDIAN ALTERPARAMTEXT
      INTEGER error
      CHARACTER*20 oldparam, newparam
      READ (9,*) oldparam
      READ (*,*) newparam
      error = ALTERPARAMTEXT (oldparam, newparam, 0, 5 )
```

CHECKLOGICALNAME Routine

The CHECKLOGICALNAME routine checks whether a saved ASSIGN message with a given logical file name exists. It also returns the message serial number associated with the saved ASSIGN message.

```
result = CHECKLOGICALNAME ( logicalname )
```

result

is an integer variable in which CHECKLOGICALNAME returns the result of the operation. See Considerations.

logicalname

is a character expression of up to 63 characters that specifies the program unit name and the logical file name in one of the following two forms:

```
programunit.filename


*.filename
```

If the logical file name does not include the program unit name component, specify only the file name. In either case logicalname cannot have leading spaces. Any character including and following a blank is ignored.

Considerations

Values returned by CHECKLOGICALNAME:

Return Value	Meaning
<0	The negated message number of a previously saved ASSIGN message with a logical file name that conflicts with the one supplied. The logical file names conflict if only one is qualified or if one is qualified by “*” and the other is qualified by a program name. If there are more than one conflicting saved ASSIGN messages, the negative of the message number of the first ASSIGN message located is returned.
0	An ASSIGN message containing the specified logical file name does not exist.
>0	Is the message number of the saved ASSIGN message with the logical file name that matches the one supplied.

Example

```
?GUARDIAN CHECKLOGICALNAME
      INTEGER error
      error = CHECKLOGICALNAME ('*.programs')
```

CHECKMESSAGE Routine

The CHECKMESSAGE routine determines whether a specific message exists or reports the number of the highest-numbered saved ASSIGN message.

```
result = CHECKMESSAGE ( messagenumber )
```

result

is an integer variable in which CHECKMESSAGE stores its result. See Considerations.

messagenumber

is an integer expression that identifies the specific ASSIGN message to check. The value for *messagenumber* must be a positive integer, 0, -1 or -3.

Considerations

- Saved ASSIGN messages

  Each saved ASSIGN message is identified by a positive integer. *messagenumber* refers to the integer value given to the specific message when it was saved. The set of numbers associated with the ASSIGN messages are integers from 1 to n, where n is normally the greatest number of ASSIGN messages saved during the initial process creation. You can specify the following values for the CHECKMESSAGE routine:

Value	Meaning
-3	Checks for the presence of a saved PARAM message
-1	Checks for the presence of a saved startup message
0	Returns the highest message number in the set of saved ASSIGN messages
>0	Checks for the presence of a saved ASSIGN message with the same number as the message supplied

- Values returned by CHECKMESSAGE

  If *messagenumber* is zero, *result* is zero if there are no saved ASSIGN messages. Otherwise, *result* is the number of the highest-numbered saved ASSIGN message.

If *messagenumber* is nonzero, *result* is zero if the specified message number does not exist, or is the value associated with *messagenumber* if the message does exist.

Example

```
?GUARDIAN CHECKMESSAGE
      INTEGER result
      result = CHECKMESSAGE ( 15 )
```

CREATEPROCESS Routine

The CREATEPROCESS routine starts a new process according to user-specified parameters. It can also send process creation messages to the new process according to the option parameter specifications.

```
result = CREATEPROCESS ( programfile, processname,
   option, priority, processor, memory, processid )
```

result

is an integer variable in which CREATEPROCESS returns the result of the operation. See Considerations on page 15-34.

programfile

is a character expression whose value is the program file name for the new process.

processname

is a character expression whose value is the name of the new process. Normally, the first six characters are significant and must be a standard Guardian process name: the first character must be a dollar sign ($), the second character must be alphabetic and any remaining characters must be alphanumeric. The minimum number of characters allowed is two, including the dollar sign. The value can be all blanks to indicate that no name is supplied.

option

is an integer expression that indicates which process creation messages are required by the new process. Its value must be 0, 1, 2, 3 or 8, 9, 10, 11. See Considerations on page 15-34.

priority

is an integer expression that specifies the priority of the new process. If its value is zero, the new process has the same priority as the creator process.

processor

is an integer expression that specifies the processor (0 to 15) in which to run the new process. If *processor* is negative, the new process executes in the same processor as the creator process.

memory

is an integer expression that specifies how many memory pages to allocate for the new process. If memory is zero, the number of *memory* pages in the new process is the value specified in the program file from which the new process is created.

processid

is an INTEGER*2 array with at least four elements that contains the process identification of the new process.

Considerations

- Values returned by CREATEPROCESS

Return Value	Meaning
0	Successful creation.
1	Required parameter is missing or illegal.
2	Illegal program file name.
3	Input file name, output file name, or default volume name cannot be converted to network form.
10–255	Guardian file-system error.
256–	NEWPROCESS error.

- Option values for CREATEPROCESS

Option Value	Meaning
0	Sends copies of all saved ASSIGN, PARAM, and startup messages from the creator process. If no saved startup message exists, a standard message is sent which specifies that the volume and subvolume, IN-file, and OUT-file are those of the creator process and the message parameter string is a null string.
1	Sends a copy of the saved startup message from the creator process. If no saved startup message exists, a standard message is sent as described above.
2	Sends a standard startup message.
3	Sends no messages; the new process is not opened by the creator process.

Normally, if a class one error (undefined externals) occurs while FORTRAN attempts to create the new process, the new process is stopped. An application can force creation of the new process by adding 8 to the normal option value (8

instead of 0, 9 instead of 1, ... 11 instead of 3). CREATEPROCESS returns the class one error result although the new process is created. The value 8 indicates that undefined externals are not considered an error.

- Opening an unnamed process

  If you omit the *processname* parameter or if it is all blanks, the system creates a process identification in timestamp format, and CREATEPROCESS returns this in its *processid* parameter if present. However, this is an integer array, not a character string, and therefore cannot be used in the FILE specifier of a FORTRAN OPEN statement. It can be used as the file name argument to a Guardian procedure that opens files, but if you want to use FORTRAN I/O statements to send messages to the new process via its $RECEIVE file, you must supply a *processname* parameter when you call CREATEPROCESS and then use the same character value in a FILE specifier in the OPEN statement for the new process.

Example

```
?GUARDIAN CREATEPROCESS
      INTEGER created
      INTEGER*2 newid(4)
      CHARACTER*6 newprocess
      newprocess = '$newpr'
      created = CREATEPROCESS (games, newprocess, 1, 0,
                               & -5, 0, newid)
```

DELETEASSIGN Routine

The DELETEASSIGN routine deletes a part or all of an ASSIGN message.

result = DELETEASSIGN ({ '*ALL*' / *portion* }, *cplist*, *messagenumber*)

result

is an integer variable in which DELETEASSIGN returns the result of the operation. See Considerations.

portion

is a character expression that identifies the particular part of the message to delete. The string value must be a legal parameter value defined for the ASSIGN message. The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

You cannot delete the message portion value associated with LOGICALNAME.

`'*ALL*'`

specifies that FORTRAN delete the entire ASSIGN message.

`cplist`

is a checkpoint list in which DELETEASSIGN records the changes to the message storage data space. DELETEASSIGN uses a maximum of three elements in *`cplist`*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

`messagenumber`

is an integer expression that identifies a specific ASSIGN message to delete. *`messagenumber`* must be a positive integer, 0, -1, or -3.

Considerations

Values returned by DELETEASSIGN

Return Value	Meaning
0	The specified ASSIGN message or message portion is deleted.
-1	A failure occurred due to a logic error. Nothing is deleted. Possible logic errors are: ● *`portion`* is not correct or does not identify a part of the ASSIGN message that can be deleted. ● *`messagenumber`* is not a positive integer. ● The contents of the checkpoint list parameter are inconsistent.
-2	Insufficient checkpoint list space is available to complete deletion; the message is unchanged.

Example

```
?GUARDIAN DELETEASSIGN
      INTEGER error
      error = DELETEASSIGN ('RECSIZE', 0, j)
```

DELETEPARAM Routine

The DELETEPARAM routine deletes specific parts of or an entire PARAM message.

```
result = DELETEPARAM ( { '*ALL*' }, cplist )
                     { portion }
```

result

is an integer variable in which DELETEPARAM returns the result of the operation. See Considerations.

portion

is a character expression that identifies the particular part of the message to delete. The string value must be a legal parameter name defined for the PARAM message. The first unused character position, if any, in *portion* must be a blank. Any characters including and following a blank are ignored.

'*ALL*'

specifies that FORTRAN delete the entire PARAM message.

cplist

is a checkpoint list in which DELETEPARAM records the changes to the message storage data space. DELETEPARAM uses a maximum of three elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

Considerations

Values returned by DELETEPARAM

Return Value	Meaning
0	The specified PARAM message or message portion is deleted.
-1	A failure occurred due to a logic error. The message is not deleted. Possible logic errors are: • The portion value is not correct. • The contents of the checkpoint list parameter are inconsistent.
-2	Insufficient checkpoint list space is available to complete the deletion. The message is unchanged.

Example

```
?GUARDIAN DELETEPARAM
      INTEGER error
      error = DELETEPARAM ('*ALL*', 0)
```

DELETESTARTUP Routine

The DELETESTARTUP routine deletes an entire startup message.

```
result = DELETESTARTUP ( '*ALL*' , cplist )
```

result

is an integer variable in which DELETESTARTUP returns the result of the operation. See Considerations.

'*ALL*'

specifies that the entire startup message is to be deleted.

cplist

is a checkpoint list in which DELETESTARTUP records the changes to the message storage data space. DELETESTARTUP uses a maximum of three elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

Considerations

Values returned by DELETESTARTUP

Return Value	Meaning
0	The message is deleted.
-1	Indicates a failure due to a logic error. Nothing is deleted. Possible logic errors are: ● The portion string value is not correct. ● The contents of the checkpoint list parameter are inconsistent.
-2	Insufficient checkpoint list space is available to complete the deletion. The message is unchanged.

Example

```
?GUARDIAN DELETESTARTUP
      INTEGER error
      error = DELETESTARTUP ( '*ALL*' , 0)
```

GETASSIGNTEXT Routine

The GETASSIGNTEXT routine retrieves a specified part of an ASSIGN message as text and assigns it to a string variable. The assignment is done according to the FORTRAN rules for a CHARACTER variable assignment.

```
result = GETASSIGNTEXT ( portion, text, messagenumber )
```

result

is an integer variable in which GETASSIGNTEXT returns the result of the operation. See Considerations on page 15-40.

portion

is a character expression with a value of 'LOGICALNAME', 'TANDEMNAME' or '*ALL*' that specifies the part of the ASSIGN message to retrieve. The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

You can specify '*ALL*' to retrieve the entire ASSIGN message.

text

is a character variable into which the retrieved message text is placed.

If the value of *portion* is 'LOGICALNAME', the text contains the program unit name and the logical file name, formatted as “programunit.filename”, and can have a maximum of 63 characters. If the logical name part does not include the program unit name component, the *text* is just “filename”.

If the value of *portion* is 'TANDEMNAME', the text is the HP file name and can have a maximum of 34 characters. The name can be all blanks.

If the value of *portion* is '*ALL*', the text contains the entire ASSIGN message. The maximum length of *text* in this case is 108 characters. Note that *text* can be a RECORD name, which FORTRAN treats as a character variable with a length equal to the sum of the length of its components.

messagenumber

is an integer constant, variable, or expression that identifies the specific ASSIGN message from which to retrieve text. *messagenumber* must be a positive integer.

Considerations

- Values returned by GETASSIGNTEXT

Return Value	Meaning
≥0	A non-negative value (zero or a positive integer) indicates a string of that many characters, before truncation or padding, is returned to text. GETASSIGNTEXT returns zero if the file name is all blanks and you request TANDEMNAME.
-1	Indicates a failure due to a logic error. Nothing is returned in *text*. Possible logic errors are: • *messagenumber* is not a positive integer. • The specified message does not exist. • LOGICALNAME or TANDEMNAME does not identify the defined text part of an ASSIGN message, or identifies a portion that does not exist in the specified message.

- If you do not name GETASSIGNTEXT in a GUARDIAN directive, you must enclose *messagenumber* in back slashes so that it will be passed by value.

Example

```
?GUARDIAN GETASSIGNTEXT
      INTEGER error
      CHARACTER *63 new name
      error = GETASSIGNTEXT ('LOGICALNAME', new name, 2)
```

GETASSIGNVALUE Routine

The GETASSIGNVALUE routine retrieves a specified part of an ASSIGN message as an integer and assigns it to a numeric variable.

```
result = GETASSIGNVALUE ( portion, val, messagenumber )
```

result

is an integer variable in which GETASSIGNVALUE returns the result of the operation. See Considerations on page 15-41.

portion

is a character expression that identifies the particular part of the message to retrieve. The string value must be the name of an ASSIGN message portion that has an integer value. The first unused character position in *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

'LOGICALNAME' and 'TANDEMNAME' are not acceptable portion values for GETASSIGNVALUE.

val

is the integer variable into which the retrieved value is placed.

messagenumber

is an integer expression that identifies a specific ASSIGN message from which GETASSIGNVALUE retrieves values. *messagenumber* must be a positive integer.

Considerations

Values returned by GETASSIGNVALUE

Return Value	Meaning
0	The specified portion value is returned to val.
-1	Indicates a failure due to a logic error; nothing is returned to val. Possible logic errors are: ● The message number is not a positive integer. ● The specified message does not exist. ● The portion parameter is not correct, does not identify a defined integer part of the ASSIGN message, or identifies a part that does not exist in the specified message.

Example

```
?GUARDIAN GETASSIGNVALUE
      INTEGER error, value
      error = GETASSIGNVALUE ('RECSIZE', value, 12 )
```

GETBACKUPCPU Routine

The GETBACKUPCPU routine retrieves a backup CPU number from the PARAM message of a FORTRAN program. This routine's operation depends on a saved PARAM message with a parameter name BACKUPCPU having a digit string value. GETBACKUPCPU is especially intended for PATHWAY users.

```
result = GETBACKUPCPU ()
```

result

is an integer variable in which GETBACKUPCPU returns the result of the operation. If a BACKUPCPU parameter exists and its value is an integer in the range 0 through 98, expressed in decimal, that value is returned. Otherwise, the integer value 99 is returned.

Example

```
?GUARDIAN GETBACKUPCPU
      INTEGER existcpu
      existcpu = GETBACKCPU ()
```

GETPARAMTEXT Routine

The GETPARAMTEXT routine obtains a specified part of the PARAM message as text and assigns it to a string variable. The assignment is done according to the FORTRAN rules for a CHARACTER variable assignment.

```
result = GETPARAMTEXT ( portion, text )
```

result

is an integer variable in which GETPARAMTEXT returns the result of the operation. See Considerations.

portion

is a character expression that identifies the particular part of the message to retrieve. The string value must be a legal parameter name for the PARAM message. The first unused character position in *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

You can specify '*ALL*' for *portion* to obtain the entire PARAM message. See Considerations.

text

is the character variable or RECORD into which the retrieved message text is placed. The text returned from a specific PARAM message parameter is the value associated with that parameter name.

Considerations

- Values returned by GETPARAMTEXT

Return Value	Meaning
≥0	Indicates a string of *result* characters, before truncation or padding, is returned to *text*. If the parameter value is a null string, a zero is returned.
-1	Indicates a failure due to a logic error, nothing is returned in *text*. Possible logic errors are: • The PARAM message does not exist. • *portion* is not correct or identifies a parameter that does not exist in the PARAM message.

- PARAM message length

  The maximum size of the PARAM message is 1028 bytes, but a character variable in FORTRAN cannot have more than 255 characters. If the PARAM message is more than 255 characters, you can declare text as a RECORD, as shown in the following example:

```
RECORD text
   INTEGER*2 msgcode, numparams
   RECORD param (0: 1023)
      CHARACTER*1 byte
   END RECORD
END RECORD
```

  If the FORTRAN program allocates a text variable or RECORD that is smaller than the PARAM message read, the PARAM message is truncated. If the program allocates a text larger than the PARAM message, trailing blanks are provided.

Example

```
?GUARDIAN GETPARAMTEXT
      INTEGER error
      CHARACTER * 20 string, oldparam
      READ (*,*) string
      error = GETPARAMTEXT (string, oldparam)
```

GETSTARTUPTEXT Routine

The GETSTARTUPTEXT routine obtains a specified part of the startup message as text and assigns it to a string variable. The assignment is done according to the FORTRAN rules for CHARACTER assignment.

```
result = GETSTARTUPTEXT ( portion , text )
```

result

is an integer variable in which GETSTARTUPTEXT returns the result of the operation. See Considerations on page 15-44.

portion

is a character expression with a value of 'VOLUME', 'IN', 'OUT', or 'STRING' that identifies the particular part of the message to retrieve.

The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

text

is a character variable or RECORD into which the retrieved message text is placed. See Considerations.

Considerations

- Values returned by GETSTARTUPTEXT

Return Value	Meaning
≥0	A non-negative value (zero or positive integer) indicates a string of that many characters, before truncation or padding, is returned to *text*. For IN or OUT, if the file name is all blanks, a zero is returned. For STRING, if the value is a null string, a zero is returned.
-1	Indicates a failure due to a logic error; nothing is assigned to text. Possible logic errors are: ● The startup message does not exist. ● The portion value is not correct or does not identify a defined part of the startup message.

- Text values for GETSTARTUPTEXT

  If you specify 'VOLUME', text contains the default node, volume, and subvolume names, formatted as

  ```
  \ node.$ volume. subvol
  ```

  and can have a maximum of 25 characters. If the volume part does not include the node name component, *text* contains $*volume. subvol*.

  If you specify 'IN', *text* is the input file name and can have a maximum of 34 characters. The name can be all blanks.

  If you specify 'OUT', *text* is the output file name and can have a maximum of 34 characters. The name can be all blanks.

  If you specify 'STRING', *text* is the startup message's parameter string, not including trailing null characters. It can be a maximum of 526 characters. You can use a RECORD name as text to circumvent the limit of 255 characters for a character variable.

Example

```
?GUARDIAN GETSTARTUPTEXT
      INTEGER error
      CHARACTER * 25 old volume
      error = GETSTARTUPTEXT ('VOLUME', old volume)
```

PUTASSIGNTEXT Routine

The PUTASSIGNTEXT routine creates or replaces a specified text part of an ASSIGN message with text obtained from a string variable.

```
result = PUTASSIGNTEXT( portion, text, cplist, messagenumber)
```

result

is an integer variable in which PUTASSIGNTEXT returns the result of the operation. See Considerations.

portion

is a character expression with a value of 'LOGICALNAME' or 'TANDEMNAME' that identifies the particular part of the message to replace. The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

text

is a character data item containing the replacement text for the ASSIGN message. See Considerations.

cplist

is a checkpoint list in which PUTASSIGNTEXT records the changes to the message storage data space. PUTASSIGNTEXT uses a maximum of six elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

messagenumber

is an integer expression that specifies the ASSIGN message into which the new text is placed. *messagenumber* must be a positive integer.

Considerations

- Results returned by PUTASSIGNTEXT

  PUTASSIGNTEXT returns an integer value that indicates the result of the operation. If the specified ASSIGN message does not exist, PUTASSIGNTEXT attempts to create one containing the supplied portion for text and default values for all other message parts. PUTASSIGNTEXT returns -1, -2, or -3 if the operation fails.

The following table shows the meaning of the integer return values:

Return Value	Meaning
≥0	A string of that many characters is assigned as the new message part value. For TANDEMNAME, PUTASSIGNTEXT returns zero if *text* is a null string.
-1	A failure due to a logic error. The message is unchanged. Possible logic errors are: ● The portion value is not correct or does not identify a defined text part of the ASSIGN message. ● The text string is not acceptable: either the logical file name or the HP file name is incorrect. ● *messagenumber* is not a positive integer. ● The requested logical file name conflicts with the logical file name of another saved ASSIGN message. ● The contents of the checkpoint list parameter are inconsistent. ● The specified ASSIGN message does not exist and the portion parameter is not LOGICALNAME.
-2	Insufficient checkpoint list space to complete the operation. The message is unchanged.
-3	Insufficient message storage space to complete the operation. The message is unchanged.

- The ASSIGN message creation succeeds if space is available for both *text* and the checkpoint message, the requested portion is the logical name, and the requested logical file name does not conflict with that of another saved ASSIGN message. All other parts are marked “not present in this message”.
- Text values returned for PUTASSIGNTEXT

  If you specify 'LOGICALNAME', *text* contains the program unit name and the logical file name, formatted as “programunit.filename”, and can have a maximum of 63 characters. If the logical name part does not include the program unit name component, the text is just “filename”.

  If you specify 'TANDEMNAME', *text* contains the HP file name and can have a maximum of 34 characters. The name can be all blanks. If it includes a node name, it must be a node known to the node on which your program runs.

Example

```
?GUARDIAN PUTASSIGNTEXT
      INTEGER error
      CHARACTER* 20 new name
      READ (*,*) new name
      error = PUTASSIGNTEXT ('TANDEMNAME', new name, 0, 10)
```

PUTASSIGNVALUE Routine

The PUTASSIGNVALUE routine creates or replaces a specified numeric part of an ASSIGN message with the value obtained from an integer variable.

```
result = PUTASSIGNVALUE( portion, value, cplist,
messagenumber )
```

result

is an integer variable in which PUTASSIGNVALUE returns the result of the operation. See Considerations on page 15-48.

portion

is a character expression that identifies the particular part of the message to replace. The string value must be the name of an ASSIGN message portion that has an integer value. The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

'LOGICALNAME' and 'TANDEMNAME' are not acceptable portion values for PUTASSIGNVALUE.

value

is an integer expression whose value PUTASSIGNVALUE stores in the ASSIGN message.

cplist

is a checkpoint list in which PUTASSIGNVALUE records the changes to the message storage data space. PUTASSIGNVALUE uses a maximum of six elements in *cplist*.For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

messagenumber

is an integer expression that identifies a specific ASSIGN message into which the new value is placed. *messagenumber* must be a positive integer.

Considerations

Results returned by PUTASSIGNVALUE

Return Value	Meaning
0	*value* is assigned as the new value of the requested message portion.
-1	Indicates a failure due to a logic error. The message is unchanged. Possible logic errors are: ● The portion value is not correct or does not identify a defined integer part of the ASSIGN message. ● The integer value supplied is not acceptable for the specified part of the ASSIGN message. ● The *messagenumber* is not a positive integer. ● The specified message does not exist. A non-existent message must be created by the PUTASSIGNTEXT routine. ● The contents of *cplist* are inconsistent.
-2	Insufficient checkpoint list space to complete the operation. The message is unchanged.
-3	Insufficient message storage space to complete the operation. The message is unchanged.

Example

```
?GUARDIAN PUTASSIGNVALUE
      INTEGER error, new value
      READ (*,*) new value
      error = PUTASSIGNVALUE ('SECEXT', new value , 0, 3)
```

PUTPARAMTEXT Routine

The PUTPARAMTEXT routine creates or replaces a specified part of the PARAM message with text obtained from a string variable.

```
result = PUTPARAMTEXT ( portion , text , cplist )
```

result

is an integer variable in which PUTPARAMTEXT returns the result of the operation. See Considerations on page 15-49.

portion

is a character expression that identifies the particular part of the message to replace. The string value must be a legal parameter name for the PARAM

message. The first unused character position of *portion*, if any, must be a blank. Any characters including and following a blank are ignored.

text

is a character variable or RECORD containing the replacement text for the PARAM message.

cplist

is a checkpoint list in which PUTPARAMTEXT records the changes to the message storage data space. PUTPARAMTEXT uses a maximum of six elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

Considerations

- If the PARAM message does not exist, PUTPARAMTEXT attempts to create one containing the supplied parameter name and value. The PARAM message creation is successful if both message space and *cplist* space are available. The new message contains only the parameter name and value supplied to PUTPARAMTEXT.
- Values returned by PUTPARAMTEXT

Return Value	Meaning
≥0	Indicates a string of that many characters is replaced as the new parameter value. A zero is returned if the parameter value supplied is all blanks.
-1	A failure occurred due to a logic error. The message is unchanged. Possible logic errors are: ● The portion value is not correct. ● he new parameter value string exceeds 255 characters. ● The total length of the new PARAM message exceeds the maximum. ● The contents of *cplist* are inconsistent.
-2	Insufficient checkpoint list space to complete the operation. The message is unchanged.
-3	Insufficient message storage space to complete the operation. The message is unchanged.

Example

```
?GUARDIAN PUTPARAMTEXT
      INTEGER error
      CHARACTER*100 new text, old text
      READ (9,*) new text
      error = PUTPARAMTEXT (old text, new text, 0 )
```

PUTSTARTUPTEXT Routine

The PUTSTARTUPTEXT routine creates or replaces a specified part of the startup message with text obtained from a string variable.

```
result = PUTSTARTUPTEXT ( portion , text , cplist )
```

result

is an integer variable in which PUTSTARTUPTEXT returns the result of the operation. See Considerations.

portion

is a character expression that identifies the particular part of the message to replace. The string value must be a legal parameter name defined for the startup message. The first unused character position of *portion*, if any, must be a blank. Any character including and following a blank is ignored.

text

is a character variable or RECORD containing the replacement text for the startup message. See Considerations.

cplist

is a checkpoint list in which PUTSTARTUPTEXT records the changes to the message storage data space. PUTSTARTUPTEXT uses a maximum of six elements in *cplist*. For additional detail, see the Checkpoint Considerations for Saved Message Utility Routines on page 15-28.

Considerations

- If the startup message does not exist, PUTSTARTUPTEXT attempts to create one containing the requested text portion and default values for all other message parts.
- Text values for PUTSTARTUPTEXT

  If you specify 'VOLUME', *text* contains the default node, volume, and subvolume names, formatted as:

  ```
  \node.$volume.subvol
  ```

  and contains a maximum of 25 characters. If the volume part does not include the node name component, *text* is just $volume.subvol.

  If you specify 'IN', *text* contains the input file name and can have a maximum of 34 characters. The file name can be all blanks.

  If you specify 'OUT', *text* is the output file name and can have a maximum of 34 characters. The file name can be all blanks.

  If you specify 'STRING', *text* contains the startup message's parameter string, not including trailing null characters, and can have a maximum of 526 characters. You can use a RECORD name for *text* to circumvent the limit of 255 characters for the length of a character variable.

- Results returned for PUTSTARTUPTEXT

Return Value	Meaning
≥0	A non-negative value (zero or positive integer) indicates a string of that many characters is replaced as the new message part value. For IN or OUT, if the file name supplied is all blanks, a zero is returned. For STRING, if the value supplied is all blanks, a zero is returned.
-1	Indicates a failure due to a logic error. The message is unchanged. Possible logic errors are: • The portion value is not correct or does not identify a defined part of the startup message. • The text string value is not acceptable The node name, volume name, subvolume name, or file name is illegal; or you specified an unknown node name or a message parameter string exceeds 526 characters. • The contents of *cplist* are inconsistent.
-2	Indicates insufficient checkpoint list space to complete the operation. The message is unchanged.
-3	Indicates insufficient message storage space to complete the operation. The message is unchanged.

The startup message creation is successful if both message space and cplist space are available. The new message contains the message part assigned. All other message parts are set to their default values. The default values for VOLUME, IN, and OUT are taken from those of the current program. The default value for the message parameter string is a null string.

Example

```
?GUARDIAN PUTSTARTUPTEXT
      INTEGER error
      CHARACTER * 8 outfile
      READ (*, *) outfile
      error = PUTSTARTUPTEXT ('OUT', outfile, 0)
```

16 Fault-Tolerant Programming

Topics covered in this section include:

When you design an application, you must decide which processes in the application need to be fault tolerant. A fault-tolerant process actually consists of two processes: a primary process and a backup process.

If the primary process fails or the processor in which it is running fails, the backup process takes over the tasks being performed by the primary process and continues running your program.

Fault-tolerant processes are said to be either NonStop processes or persistent processes. Both NonStop processes and persistent processes are implemented as pairs of Guardian processes. FORTRAN treats all process pairs as NonStop processes.

For each process pair, one process is designated as the primary process, the other as the backup process. During normal processing, the primary process performs all the tasks for your program. The backup process does not duplicate the work of the primary process. System routines, however, transfer information specified by the primary process to the data areas of the backup process.

A NonStop process is designed such that if its primary process fails, its backup process has enough of the state of the primary process that the backup process can continue running your program with minimal impact on a user at a terminal. Even data at a terminal might be retained in the terminal itself.

A persistent process also runs as a process pair but the primary process sends the backup process only enough information to enable the backup process to take over processing if the primary process fails. The backup process, however, does not hold all the state of the primary process. You might want to use persistent processes if:

- You are depending on TMF to protect your transactions and you do not need a backup process to maintain the integrity of your application.
- You do not want any processor time given to managing a backup process, however minimal the extra time might be.
- You want to ensure that the application is always available at a terminal. For example, if you run a terminal without a command interpreter, a persistent process ensures that you always have an application screen displayed on your terminal,

even if the primary process fails. You might, however, have to reenter data, reinitialize your environment, and so forth.

To run a fault-tolerant process, your program must:

- Be a named process
- Execute a START BACKUP statement to create a backup process
- Execute CHECKPOINT statements to send data to the backup process
- Specify the NONSTOP directive if you specify the ENV COMMON directive
- Not specify the NONSTOP OFF TACL PARAM if you compiled your program with ENV COMMON

The FORTRAN run-time library performs all other tasks needed to support fault-tolerant operation for your process.

Assigning a Process Name

A fault-tolerant program consists of two separate processes (both created from the same object program) running in two separate processors. Both the primary process and the backup process have the same process name.

You can assign a process name to a process by:

- Specifying the run-time NAME option with a five-character name preceded by a dollar ($) sign; for example:

  ```
  1> RUN program/NAME $nsprc/
  ```

- Using a system-assigned name by specifying the NAME run-option without giving a specific name; for example:

  ```
  1> RUN program/IN file1, OUT file2, NAME/
  ```

- Specifying the RUNNAMED compiler directive:

  ```
  ?RUNNAMED
  ```

- Using the D-series Binder to set the RUNNAMED attribute in an object file:

  ```
  SET RUNNAMED ON
  ```

For more information about the RUNNAMED compiler directive, see Section 10, Compiler Directives.

For more information about Binder support for the RUNNAMED attribute, see the *Binder Manual*.

Processes

A program is a static set of instruction codes and initialized data. It can be represented as source statements, such as a FORTRAN program, or as an object program or a run unit consisting of machine instructions and initialized data. A process is the constantly changing states of a running program. You can run multiple copies of the same program file concurrently. Each execution of the program constitutes a separate process.

A process consists of:

- A code area that contains the instruction codes to execute; all processes that execute the same program file in the same processor share the same code area.
- A data area that contains the program's variables and temporary storage; each process has its own data area.
- A process ID assigned by the operating system.
- A process control block (PCB), which is used by the operating system to control execution; the PCB contains pointers to the code and data areas, information on the current status of the process, and pointers to files opened by the process.

Process Pairs

A process can recover from any hardware failure except a failure of the processor in which it is running. Therefore, a fault-tolerant process consists of a primary process and a backup process—called a process pair.

A process pair consists of two executions of the same object program. The primary process runs in one processor and the backup process runs in another.

Checkpoint messages, sent periodically from the primary process to the backup process, keep the backup process informed of the status of the primary process and the data with which it is operating. If the backup process receives a system message notifying it of the failure of its primary process or the processor in which the primary process was running, the backup process takes over the role of the primary process and continues executing instructions, beginning with the FORTRAN statement that follows the most recent checkpoint that established a takeover point. (All CHECKPOINT statements do not establish a takeover point.)

Overview of Fault- Tolerant Programs

The following actions occur when you run a fault-tolerant program:

- The primary process opens the initial set of files required for its operation.
- The primary process starts its backup process in another processor by executing a START BACKUP statement. START BACKUP, in addition to starting the backup process, sends the backup checkpoint information for files open in the primary process. Process pairs open files in a way that permits both members of the pair to access the file. For disk files opened in this way, a record lock or file lock specified by the primary process is equivalent to a lock by the backup.
- The backup process, at the start of its execution, automatically begins monitoring the primary process. The backup proceeds no further unless a failure occurs.
- The primary process begins executing its main processing loop. At critical points in the loop (for example, just before write operations to disk files), the primary process executes CHECKPOINT statements to send program state and file control data to the backup process and establish takeover points for the backup. A takeover point is established in the backup process by the most recently executed CHECKPOINT statement that does not specify STACK='NO'. OPEN and CLOSE statements also establish takeover points in the backup unless you specify STACK = 'NO' for those statements.

  A program can contain many CHECKPOINT statements. You usually code CHECKPOINT statements so as to ensure that logical groupings of data are preserved in the backup process.

  For example, you frequently execute a CHECKPOINT statement immediately before you execute a WRITE statement so that if the WRITE statement fails, or the processor in which your primary runs fails, all the processing up to the point of the WRITE statement is preserved in the backup process. If the backup process takes over processing, the first statement it executes is the WRITE statement for which it has all the information it needs. Here is an example:

  Primary process:

```
...
CHECKPOINT
WRITE(6, 100) r, s
```

  Primary's processor fails, backup takes over:

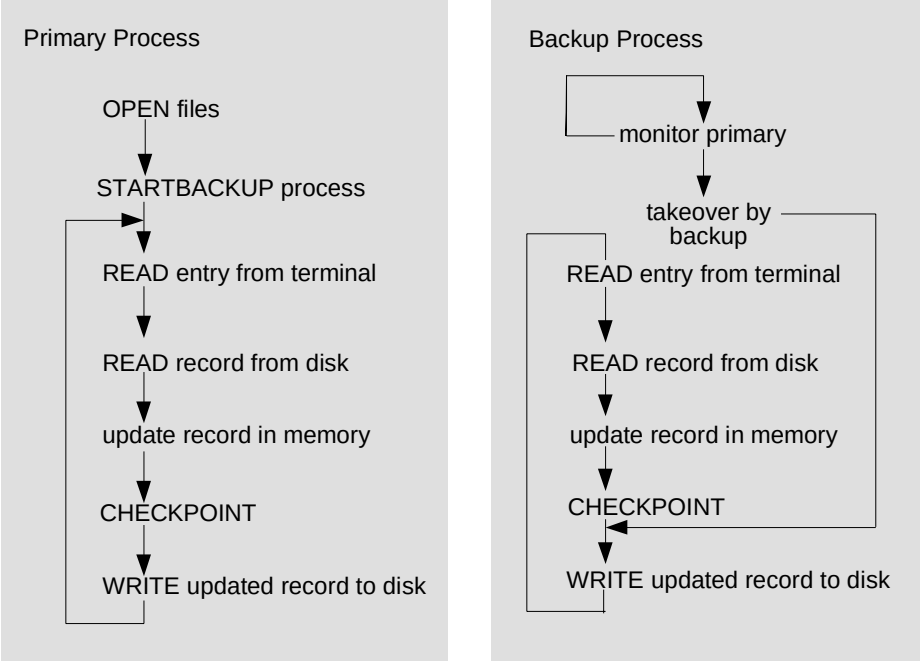
```
CHECKPOINT              <-- Backup does NOT re-execute
WRITE(6, 100) r, s      <-- Backup begins HERE by re-
                            executing the WRITE statement
```

- The START BACKUP statement does not establish a takeover point. You must execute a CHECKPOINT statement that does not specify STACK = 'NO' to establish a takeover point.
- If the primary process, or the processor in which it runs, fails, the backup process begins executing instructions at the most recent takeover point. (Takeover points are described earlier within this subsection.) The backup process becomes the new primary process.
- In some cases, FORTRAN starts a new backup process in the former primary's processor. Under some circumstances, however, FORTRAN either does not or cannot start a new backup process. For example, if the processor in which the primary process was running is down, FORTRAN cannot start a new backup process in that processor and returns backup status 102. If the primary process has failed more than ten times, FORTRAN does not start a new backup process. Table 7-10 in Section 7, Statements lists the backup status codes.

  If FORTRAN returns a backup status code that indicates that it has not started a new backup process, your program must execute a START BACKUP statement if you want to continue running as a NonStop process.

Figure 16-1 on page 16-6 illustrates a typical fault-tolerant application. The backup process remains in the monitor state while the primary is operational. If the primary fails, the backup leaves the monitor loop and begins executing instructions at the most recently established takeover point.

Figure 16-1. Fault-Tolerant Processing



The backup process monitors the primary process while the primary runs. If the primary cannot continue running—for example, it fails or the processor in which it is running fails—the backup process leaves the monitor state and takes over running your program by executing the instructions that immediately follow the last CHECKPOINT executed by the primary process.

VST1601.vsd

Checkpointing

In general, you need to checkpoint the following types of information:

- Individual data entities (usually file buffers, but can be any data items desired)
- File “sync blocks” (these contain control information about the current status of a disk file—for example, the current values of the file pointers—or a process file)
- RECEIVE

When a CHECKPOINT statement executes in the primary process, FORTRAN formats the information to checkpoint and sends it to the backup process in an interprocess message. The backup process automatically receives and processes the message.

FORTRAN checkpoints information to the backup process when the backup process is created, when an OPEN or CLOSE statement is executed, and when you access a file without having explicitly opened the file. FORTRAN implicitly opens a file if you reference the file in an I/O statement without having opened it. As part of the implicit open, FORTRAN checkpoints information to your backup process. The checkpoint specifies STACK = 'YES'. Therefore, an implicit open establishes a takeover point.

The location of checkpoint statements in your program depends on the requirements of your application. As a general rule, you include a CHECKPOINT statement just before a WRITE statement to a disk file or to $RECEIVE; a server process might execute a checkpoint statement after reading from $RECEIVE. It is essential to checkpoint before any nonretryable I/O operation, but which operations can be repeated and which cannot depends on the application's task and the program logic.

A retryable operation is one that can be repeated any number of times and yield the same result each time; read operations generally fall into this category. A nonretryable operation is one that cannot be trusted to yield the same result if it is repeated; write, rewrite, and delete operations generally fall into this category.

Checkpointing File Buffers

The primary purpose of checkpointing file data buffers is to give the backup process all the information it needs to reexecute an I/O request if the primary fails. Usually, data buffer checkpointing occurs just before the data is written.

You can also use data buffer checkpointing to eliminate the need for the backup process to reexecute an I/O request. Terminal input is an example of this: The data is checkpointed on receipt to reduce the chance of the operator's having to reenter it.

Checkpointing File Status Information

When a CHECKPOINT statement specifies a unit or file number, the system passes the current status of the file to the file system that is running in the backup process's processor. If you specify a SYNCDEPTH greater than zero when you open the file, the file's status includes a system-assigned unique identification number for each I/O you execute for the file. If your primary process fails and the backup process—which begins executing instructions at the previous takeover point—re-executes an already completed I/O operation, the receiving process of your I/O request does not reexecute the request but, instead, returns the same reply that it has saved from the original I/O request.

The following code is typical of a server process. The server reads a requester's message from $RECEIVE, executes one or more I/O operations to a file, and returns a reply to the requester. The server might contain statements such as the following:

```
      COMMON request_msg, db_record
      CHARACTER*132 reply_msg
      OPEN (UNIT=1,FILE='$RECEIVE')
      OPEN (UNIT=2,FILE='employee',SYNCDEPTH=5)
      START BACKUP (BACKUPSTATUS=ierr)
            ...
10    READ (UNIT=1) request_msg
      CHECKPOINT(UNIT=1, UNIT=2,STACK='YES') request_msg,
        process request, generate a DB record and a reply
      CHECKPOINT(UNIT=1, UNIT=2,STACK='YES') db_record
      WRITE (UNIT=2) db_record
      WRITE (UNIT=1) reply_msg
      GO TO 10
```

Requesters and Servers

In the preceding code, there are two separate files to consider: $RECEIVE and the disk file EMPLOYEE. The server opens both $RECEIVE and the EMPLOYEE file. Because unit 2, the disk file, specifies SYNCDEPTH = 5, the system must remember the completion status of up to five nonretryable requests for EMPLOYEE—for example, any WRITE requests.

After opening units 1 and 2, the server reads $RECEIVE to obtain REQUEST_MSG, which contains a request and any data needed by the server. Next, the server checkpoints the file status information for the files associated with units 1 and 2, the data stack from the initial stack marker to the top of the stack (STACK='YES'), and REQUEST_MSG. The server must specifically name REQUEST_MSG in the CHECKPOINT statement because REQUEST_MSG is in a common block and, therefore, is not checkpointed as part of the stack. The server then processes the request and executes a stack checkpoint that includes a disk record (DB_RECORD). The reply to the requester (REPLY_MSG) is also checkpointed because it is in the stack. Finally, the server writes the database record (DB_RECORD), writes the reply (REPLY_MSG) to $RECEIVE, and branches to the top of its read loop to read the next request.

The preceding code is easier to understand if you consider the following:

- A process, A, that opens a second process, B, and sends requests to B is, by definition, a requester.
- A process, C, that receives requests for services from another process, B, is, by definition, a server.
- A process is often both a requester and a server. In the text that follows, you will see that the preceding code is a server, for example B, to requesters, A. As a server, the preceding code receives requests from $RECEIVE. In addition, the preceding code is, itself, a requester, B, to another server, C. As a requester, the preceding code opens a disk file, EMPLOYEE, and sends requests to the disk process, which, in this context, is a server.

  There is no fundamental end to such a list. An application can have numerous processes that each, in turn, act as both requester and server.

Server processes can support fault tolerance in two senses.

- A server can run as a NonStop process so that the server can continue running, even if its primary process fails.
- A server can support NonStop requester processes, so that if the requester's primary process fails, the server correctly processes duplicate requests that it receives from the requester's backup process following the takeover by the backup.

NonStop Server Processes

When a server's primary process fails, its backup begins executing your program at the FORTRAN instruction that immediately follows the last stack checkpoint. This could be after a FORTRAN CHECKPOINT, OPEN, or CLOSE statement or after any I/O statement that implicitly opens a unit.

Managing $RECEIVE

The FORTRAN run-time library in the former backup, now the primary, recognizes that a takeover has occurred and discards each message that it has read from $RECEIVE but to which it has not yet replied. If the old primary failed before it wrote its reply to $RECEIVE, the request failed, and the file system automatically redirects the request to the new primary (assuming the server was opened with SYNCDEPTH greater than zero). If the old primary failed after it wrote its reply to $RECEIVE, that request was complete and nothing more need be done. (The file system redirects a requester's messages only a limited number of times, typically less than three.)

The new primary then loops back to read a request from $RECEIVE: either the same request that the old primary was processing when it failed or a new request.

Managing the Disk File

The new server primary might write to the disk up to five records that were already written by the former primary. Because the server specified SYNCDEPTH=5 when it opened the disk file, the disk process saves the replies to up to five write requests since the last stack checkpoint. If the disk process receives a request to which it has already responded, it returns the saved reply and does not do the actual I/O a second time.

Supporting NonStop Requester Processes

If a fault-tolerant requester fails, its backup (now the new primary) can reissue the last request. Because the server's function (adding employee records) is not retryable, the server must be prepared to recognize a duplicate request and resend the reply, rather than redoing the operation. It accomplishes this by saving replies to messages it receives from the requester. The server must save at least the number of reply messages that the requester specified as its SYNCDEPTH parameter when it opened the server. If a FORTRAN run-time library routine receives a duplicate request, it returns the same reply message that it returned the first time it replied to the message.

If a server is running as a NonStop process, it must checkpoint the saved replies to its backup process as well.

Checkpointing $RECEIVE

Because $RECEIVE is a dynamic file, naming $RECEIVE in a CHECKPOINT statement signifies nothing in itself; it does, however, signal the FORTRAN run-time support system that the next write operation via $RECEIVE will be a reply to a nonretryable request, which must be saved for use in case of primary requester failure.

Checkpointing Large Amounts of Data

The maximum size of a checkpoint message is 32,500 bytes. The amount of user data checkpointed in a checkpoint message is less than 32,500 bytes because the message includes header and control information added by the system. If your application needs to checkpoint more data than can fit in one checkpoint message, you must checkpoint the data by executing multiple CHECKPOINT statements.

If you execute more than one CHECKPOINT statement to checkpoint your data to the backup process, you must not establish a takeover point (by specifying STACK='YES') until you have sent all the data to the backup process. Otherwise, the data in the backup process might be inconsistent when a takeover occurs—that is, some of the data in the backup might be from a previous takeover point, and other data might be the data that you have just sent to the backup process.

For example, the following code shows how you might checkpoint a large array A consisting of 100,000 bytes. A is allocated in extended memory:

```
      DIMENSION A(100000)
   10 CHECKPOINT (STACK='YES') global-data       <-- Establish a
                                                     takeover pt
   15 CONTINUE
      ...
      DO 20 I=1,4
   20    CHECKPOINT (STACK='NO') part-of-array-A < -- Do not
                                                     establish a
                                                     takeover pt
   30 CHECKPOINT (STACK='YES') global-data       <-- Establish a
                                                     takeover pt
```

Execution of the previous code proceeds as follows:

1. The CHECKPOINT statement at label 10 establishes a takeover point prior to checkpointing the array A.
2. The CHECKPOINT statement at label 20—which is in a DO loop—transmits data but does not establish a takeover point because it specifies STACK='NO'.
3. The CHECKPOINT statement labeled 30, following the DO-loop, establishes a new takeover point because it specifies STACK = 'YES'.

If the primary fails at any point after executing the CHECKPOINT statement labeled 10 but before executing the CHECKPOINT statement labeled 30, the backup process takes over at the CONTINUE statement labeled 15.

If the CHECKPOINT statement at label 20 had specified STACK = 'YES' and a failure occurred before all of array A was transferred to the backup process, some of the values in array A in the backup process would be left over from a previous checkpoint of array A, but some of the values might be from the current transfer of array A.

For additional information on fault-tolerant processing, see the *Guardian Programmer's Guide*.

Starting a New Backup Process

The following list describes the possible actions of the new primary process—formerly the backup process—after a takeover from the former primary process as a result of either a failure or a call to a Guardian routine to stop the process.

- If the former primary process called STOP or PROCESS_STOP_ and the START BACKUP statement did not set bit 13 in its OPTION specifier, the backup process also stops immediately.
- If there have been more than ten takeovers by the backup process, FORTRAN does not start another backup process, and returns from the CHECKPOINT statement with BACKUPSTATUS = 5000.
- If the START BACKUP statement had bit 11 set (recreate a backup process immediately after a takeover) in its OPTION specifier, FORTRAN attempts to create a new backup process in the former primary's processor. If the takeover was not caused by a processor failure and FORTRAN cannot start a new backup process in the former primary processor, FORTRAN terminates your process. Otherwise, it returns from the CHECKPOINT statement with BACKUPSTATUS = 100 or BACKUPSTATUS = 101.
- If the START BACKUP statement did not set bit 11 in its OPTION specifier, FORTRAN allows the new primary process to run without a backup for a while, and arranges for the next CHECKPOINT statement to attempt to create a new backup process in the former primary's processor. It then returns from the present CHECKPOINT statement with BACKUPSTATUS = 100 or 101.

  If the former primary's processor failed, FORTRAN does not attempt to create a new backup process, but only returns from the CHECKPOINT statement with BACKUPSTATUS = 102.

When the FORTRAN run-time system cannot start a new backup process because the former primary's processor is down, the application program must implement one of the following strategies:

- Run without a backup for the remainder of the program's execution.
- Periodically execute a START BACKUP statement on the failed processor. This could be done every time a CHECKPOINT statement returns BACKUPSTATUS = 1000 (backup CPU down).
- Execute a START BACKUP statement specifying a different processor.

A ASCII Character Set

Table A-1. ASCII Character Set (page 1 of 4)

	Octal				
Char	**Left Byte**	**Right Byte**	**Hex**	**Dec**	**Meaning**
NUL	000000	000000	00	0	Null
SOH	000400	000001	01	1	Start of heading
STX	001000	000002	02	2	Start of text
ETX	001400	000003	03	3	End of text
EOT	002000	000004	04	4	End of transmission
ENQ	002400	000005	05	5	Enquiry
ACK	003000	000006	06	6	Acknowledge
BEL	003400	000007	07	7	Bell
BS	004000	000010	08	8	Backspace
HT	004400	000011	09	9	Horizontal tabulation
LF	005000	000012	0A	10	Line feed
VT	005400	000013	0B	11	Vertical tabulation
FF	006000	000014	0C	12	Form feed
CR	006400	000015	0D	13	Carriage return
SO	007000	000016	0E	14	Shift out
SI	007400	000017	0F	15	Shift in
DLE	010000	000020	10	16	Data link escape
DC1	010400	000021	11	17	Device control 1
DC2	011000	000022	12	18	Device control 2
DC3	011400	000023	13	19	Device control 3
DC4	012000	000024	14	20	Device control 4
NAK	012400	000025	15	21	Negative acknowledge
SYN	013000	000026	16	22	Synchronous idle
ETB	013400	000027	17	23	End of transmission block
CAN	014000	000030	18	24	Cancel
EM	014400	000031	19	25	End of medium
SUB	015000	000032	1A	26	Substitute
ESC	015400	000033	1B	27	Escape
FS	016000	000034	1C	28	File separator
GS	016400	000035	1D	29	Group separator
RS	017000	000036	1E	30	Record separator

Table A-1. ASCII Character Set (page 2 of 4)

	Octal				
Char	**Left Byte**	**Right Byte**	**Hex**	**Dec**	**Meaning**
US	017400	000037	1F	31	Unit separator
SP	020000	000040	20	32	Space
!	020400	000041	21	33	Exclamation point
"	021000	000042	22	34	Quotation mark
#	021400	000043	23	35	Number sign
$	022000	000044	24	36	Dollar sign
%	022400	000045	25	37	Percent sign
&	023000	000046	26	38	Ampersand
'	023400	000047	27	39	Apostrophe
(	024000	000050	28	40	Opening parenthesis
)	024400	000051	29	41	Closing parenthesis
*	025000	000052	2A	42	Asterisk
+	025400	000053	2B	43	Plus
,	026000	000054	2C	44	Comma
-	026400	000055	2D	45	Hyphen (minus)
.	027000	000056	2E	46	Period (decimal point)
/	027400	000057	2F	47	Right slash
0	030000	000060	30	48	Zero
1	030400	000061	31	49	One
2	031000	000062	32	50	Two
3	031400	000063	33	51	Three
4	032000	000064	34	52	Four
5	032400	000065	35	53	Five
6	033000	000066	36	54	Six
7	033400	000067	37	55	Seven
8	034000	000070	38	56	Eight
9	034400	000071	39	57	Nine
:	035000	000072	3A	58	Colon
;	035400	000073	3B	59	Semicolon
<	036000	000074	3C	60	Less than
=	036400	000075	3D	61	Equals
>	037000	000076	3E	62	Greater than
?	037400	000077	3F	63	Question mark

Table A-1. ASCII Character Set (page 3 of 4)

	Octal				
Char	**Left Byte**	**Right Byte**	**Hex**	**Dec**	**Meaning**
@	040000	000100	40	64	Commercial at sign
A	040400	000101	41	65	Uppercase A
B	041000	000102	42	66	Uppercase B
C	041400	000103	43	67	Uppercase C
D	042000	000104	44	68	Uppercase D
E	042400	000105	45	69	Uppercase E
F	043000	000106	46	70	Uppercase F
G	043400	000107	47	71	Uppercase G
H	044000	000110	48	72	Uppercase H
I	044400	000111	49	73	Uppercase I
J	045000	000112	4A	74	Uppercase J
K	045400	000113	4B	75	Uppercase K
L	046000	000114	4C	76	Uppercase L
M	046400	000115	4D	77	Uppercase M
N	047000	000116	4E	78	Uppercase N
O	047400	000117	4F	79	Uppercase O
P	050000	000120	50	80	Uppercase P
Q	050400	000121	51	81	Uppercase Q
R	051000	000122	52	82	Uppercase R
S	051400	000123	53	83	Uppercase S
T	052000	000124	54	84	Uppercase T
U	052400	000125	55	85	Uppercase U
V	053000	000126	56	86	Uppercase V
W	053400	000127	57	87	Uppercase W
X	054000	000130	58	88	Uppercase X
Y	054400	000131	59	89	Uppercase Y
Z	055000	000132	5A	90	Uppercase Z
[	055400	000133	5B	91	Opening bracket
\	056000	000134	5C	92	Back slash
]	056400	000135	5D	93	Closing bracket
^	057000	000136	5E	94	Circumflex
_	057400	000137	5F	95	Underscore
`	060000	000140	60	96	Grave accent

Table A-1. ASCII Character Set (page 4 of 4)

	Octal				
Char	**Left Byte**	**Right Byte**	**Hex**	**Dec**	**Meaning**
a	060400	000141	61	97	Lowercase a
b	061000	000142	62	98	Lowercase b
c	061400	000143	63	99	Lowercase c
d	062000	000144	64	100	Lowercase d
e	062400	000145	65	101	Lowercase e
f	063000	000146	66	102	Lowercase f
g	063400	000147	67	103	Lowercase g
h	064000	000150	68	104	Lowercase h
i	064400	000151	69	105	Lowercase i
j	065000	000152	6A	106	Lowercase j
k	065400	000153	6B	107	Lowercase k
l	066000	000154	6C	108	Lowercase l
m	066400	000155	6D	109	Lowercase m
n	067000	000156	6E	110	Lowercase n
o	067400	000157	6F	111	Lowercase o
p	070000	000160	70	112	Lowercase p
q	070400	000161	71	113	Lowercase q
r	071000	000162	72	114	Lowercase r
s	071400	000163	73	115	Lowercase s
t	072000	000164	74	116	Lowercase t
u	072400	000165	75	117	Lowercase u
v	073000	000166	76	118	Lowercase v
w	073400	000167	77	119	Lowercase w
x	074000	000170	78	120	Lowercase x
y	074400	000171	79	121	Lowercase y
z	075000	000172	7A	122	Lowercase z
{	075400	000173	7B	123	Opening brace
|	076000	000174	7C	124	Vertical line
}	076400	000175	7D	125	Closing brace
~	077000	000176	7E	126	Tilde
DEL	077400	000177	7F	127	Delete

B Syntax Summary

This appendix provides a syntax summary for FORTRAN Statements and Compiler Directives on page B-12.

FORTRAN Statements

This subsection specifies the syntax of all FORTRAN statements.

```
name = { arithmetic-expression  }
       { character-expression   }
       { logical-expression     }
```

Defines the value of an arithmetic, character, or logical entity.

```
ASSIGN label TO name
```

Assigns the value of a statement label to an integer variable.

```
BACKSPACE { unit                                                                                                  }
          { ( unit [, IOSTAT=ios] [, ERR=lbl] ... )                                                               }
          { ( { UNIT=unit } [, UNIT=unit ] [, IOSTAT=ios ] [, ERR=lbl ] ... ) }
          { ( { IOSTAT=ios } [, UNIT=unit ] [, IOSTAT=ios ] [, ERR=lbl ] ... ) }
          { ( { ERR=lbl   } [, UNIT=unit ] [, IOSTAT=ios ] [, ERR=lbl ] ... ) }
```

Backspaces one record in the file connected to the unit.

```
BLOCK DATA [ subprog-name ]
```

Designates the beginning of a block data subprogram.

```
CALL subroutine-name [ ( arg [, arg ]... ) ]
```

Transfers control to the specified subroutine.

```
CHARACTER [* len] name [ dimension ] [* len]
[, name [ dimension ] [* len] ]...
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

Defines a variable, array, symbolic constant, RECORD field, function, or dummy procedure as character type.

```
CHECKPOINT [([UNIT=]unit [, UNIT=unit ]... [, cpt-spec]... )
           [(cpt-spec [, cpt-spec]...)                      ]
              [ data [, data ]... ]
```

cpt-spec

is one of the following:

```
BACKUPSTATUS = status
CPLIST = checkpoint-list
ERR = label
FILENUM = exp
STACK = stack
```

Establishes a takeover point for a backup process, or transfers the data and environment information needed by a backup process or both.

```
CLOSE ( close-spec [, close-spec ]... )
```

close-spec

is one of the following:

```
[UNIT=] unit
ERR = label
IOSTAT = ios
STACK = stack
STATUS = status
```

Disconnects a file from a specified unit and specifies the status of the file after disconnection.

```
COMMON [ / [ cb ] / ] list [ [,] / [ cb ] / list ]...
```

Defines one or more areas of memory in which program units can share data.

```
COMPLEX name [ dimension ] [, name [ dimension ] ]...
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

Specifies that the symbolic name of a constant, variable, array, RECORD field, function, or dummy procedure is of type complex.

```
CONTINUE
```

The execution of the CONTINUE statement has no effect. It is normally used as the last statement of a DO loop.

```
DATA list / data / [ [,] list / data / ]...
```

Assigns initial values for variables, arrays, array elements, and substrings at compile time.

```
DIMENSION name dimension [, name dimension ]...
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

Declares an array and specifies the number of elements in each dimension of the array.

```
DO label [,] var = iexp , fexp [, incr ]
```

Defines the beginning and end of a sequence of statements to execute repeatedly.

```
DOUBLE PRECISION name [ dimension ] [, name [ dimension ]
]...
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

Specifies that the symbolic name of a constant, variable, array, RECORD field, function, or dummy procedure is double precision type.

```
ELSE
```

Identifies the end of a block of statements that was preceded by an IF or ELSE IF statement, and the beginning of a block of statements to execute if all preceding IF and ELSE IF tests were false. The block of statements introduced by the ELSE statement must be followed by an END IF statement.

```
ELSE IF
```

Identifies the end of a block of statements that was preceded by an IF or ELSE IF statement, and the beginning of a block of statements to execute if all preceding IF and ELSE IF tests were false. The block of statements introduced by the ELSE IF statement can be followed by an another ELSE IF statement or by an END IF statement.

```
END
```

Identifies the physical end of a program unit.

```
END IF
```

Identifies the end of a block of statements that was preceded by an IF, ELSE IF, or ELSE statement. END IF terminates unconditionally a sequence of statements that begins with an IF statement.

```
ENDFILE { unit                                                          }
        {([UNIT=] unit [, IOSTAT=ios] [, ERR=label ])                  }
```

Writes an end of file as the next record of the file connected to the specified unit.

```
ENTRY name [ ( [ dummy [, dummy ]... ] ) ]
```

Provides an alternate entry point to a subroutine or function subprogram and allows you to specify an alternate dummy argument list for the subprogram.

```
EQUIVALENCE ( var-list ) [, ( var-list ) ]...
```

Specifies one or more sequences of variables (*var-lists*), each of which sequences specifies two or more variables within a program unit that access the same area of memory.

```
EXTERNAL proc-name [, proc-name ]...
```

Identifies names of external procedures, thus enabling the names to be used as actual arguments to subprograms.

```
FORMAT ( [ format-list ] )
```

Specifies a format for I/O operations.

```
[ type ] FUNCTION func-name ( [ dummy [, dummy ]... ] )
```

Designates the beginning of a function subprogram.

```
function-name ( [ dummy [, dummy ]... ] ) = expression
```

Defines a statement function.

```
GO TO label
```

Executes an unconditional transfer of control to the statement labeled label.

```
GO TO ( label [, label ]... ) [,] exp
```

Executes a computed GO TO statement. Transfers control to the label whose position in the list of labels corresponds to the value of *exp*.

```
GO TO ivar [ [,] ( label [, label ]... ) ]
```

Executes an assigned GO TO statement. Transfers control to the statement whose label was assigned to *ivar* in a previously executed ASSIGN statement.

```
IF ( exp ) label1, label2, label3
```

Executes an arithmetic IF statement. Transfers control to *label1* if *exp* is less than zero, to *label2* if *exp* equals zero, to *label3* if *exp* is greater than zero.

```
IF ( exp ) statement
```

Executes a logical IF statement. If the logical expression *exp* is true, executes *statement*. Otherwise, executes the next inline instruction.

```
IF ( exp ) THEN
     if-block
   [ ELSE IF ( exp ) THEN
     if-block ]...
   [ ELSE
     if-block ]
  END IF
```

Defines a block-IF statement. Defines one or more sequences of statements that are conditionally executed based on the result of evaluating the *exp* expressions.

```
IMPLICIT type ( char-list ) [, type ( char-list ) ]...
```

Specifies the default data type associated with the first character of a variable's name.

```
INQUIRE ( { [UNIT=] unit     } [, inq-spec ]... )
          { FILE=filename    }
```

inc-spec

is one of the following:

ACCESS = *acc*	NAME = *f*
BLANK = *blnk*	NAMED = *nmd*
DIRECT = *dir*	NEXTREC = *nr*
ERR = *label*	NUMBER = *n*
EXIST = *ext*	OPENED = *open*
FORM = *form*	RECL = *reclen*
FORMATTED = *fmt*	SEQUENTIAL = *seq*
IOSTAT = *ios*	UNFORMATTED = *unf*

Ascertains the properties of a file or the properties of the connection of a specified unit.

```
{ INTEGER   }
{ INTEGER*2 } name [ dimension ] [, name [ dimension ] ]...
{ INTEGER*4 }
{ INTEGER*8 }
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

Specifies the storage size and type of a symbolic constant, variable, array, RECORD field, function, or dummy procedure name.

```
INTRINSIC function [, function ]...
```

Identifies a name as representing an intrinsic function and enables the use of an intrinsic function name as an actual argument.

```
LOGICAL name [ dimension ] [, name [ dimension ] ]...
```

dimension

is:

([*lower*:] *upper* [, [*lower*:] *upper*]...)

Defines a variable, array, symbolic constant, RECORD field, function, or dummy procedure as logical type.

```
OPEN ( [UNIT=] unit [, open-spec ]... )
```

open-spec

is one of the following:

ACCESS = *acc*	PROTECT = *protect*
BLANK = *blnk*	RECL = *recl*
ERR = *label*	SPACECONTROL = *space*
FILE = *filename*	STACK = *stack*
FORM = *form*	STATUS = *stat*
IOSTAT = *ios*	SYNCDEPTH = *sync*
MODE = *mode*	TIMED = *time*

Associates an existing file with a unit number, creates a new file and associates it with a unit number, or changes certain attributes of an existing file.

```
PARAMETER ( name = exp [, name = exp ]... )
```

Assigns a symbolic name to a constant value.

```
PAUSE [ message ]
```

Temporarily halts program execution and displays message if present.

```
POSITION ( [UNIT=] unit [, IOSTAT = ios ]
[, ERR = lbl ], position )
```

position

is a position specifier in one of the following, mutually exclusive forms:

```
REC = recno
```

or

```
KEY = key, KEYLEN = exp, KEYID = kid, MODE = mode
[, COMPARELEN = clen ] [, SKIPEXACT = skip ]
```

Enables random access of structured files, either by record number or by specified primary or alternate keys.

```
PRINT format [, output-list ]
```

Writes data to the preconnected output unit, unit 6.

```
PROGRAM program-name
```

Assigns a symbolic name to the main program unit.

```
READ { format [, input-list]                          }
     { (read-spec [, read-spec]. . . ) [input-list]   }
```

read-spec

is one of the following:

```
END = endlbl            PROMPTLENGTH = plen
ERR = lbl               REC = rec
[FMT=] format           SOURCE = receive
IOSTAT = ios            TIMEOUT = to
LENGTH = len            [UNIT=] unit
LOCK = lock             UPDATE = upd
PROMPT = message
```

Inputs data from a specified unit or file.

```
REAL name [ dimension ] [, name [ dimension ] ]...
```

dimension

is:

```
( [ lower:] upper [, [ lower:] upper ]... )
```

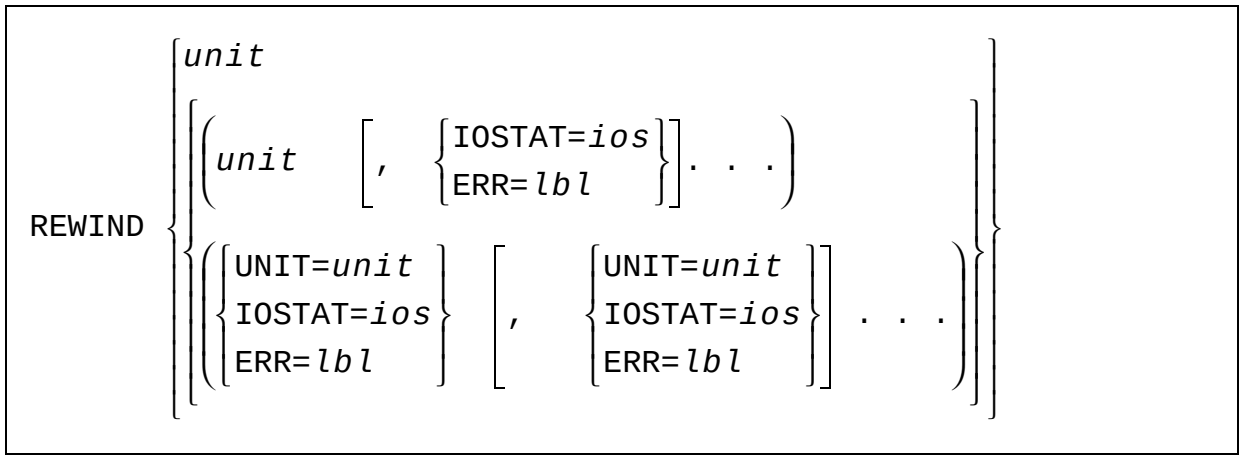
Specifies that the symbolic name of a constant, variable, array, RECORD field, function, or dummy procedure is of type real.

```
RECORD record-name [ ( [ lower:] upper) ]
[ field-declaration ]...
END RECORD
```

Defines a data structure that can include data of different types.

```
RETURN [ iexp ]
```

Terminates execution of a subprogram and returns control to the calling program unit.



Positions the file connected to the specified unit at its initial point.

```
SAVE [ name [, name ]... ]
```

Saves the status of specified entities after the termination of a subprogram.

```
START BACKUP [ ( start-spec [, start-spec ]...) ]
```

start-spec

is one of the following:

BACKUPSTATUS = *var*

CPU = *number*

ERR = *label*

OPTION = *int*

Defines control options for fault-tolerant processing. Starts backup process, sends file information to the backup process, checks file synchronization information, and checkpoints all usable memory.

```
STOP [ message ]
```

Terminates program execution and displays message if present.

```
SUBROUTINE name [ ( dummy [, dummy ]... ) ]
```

Identifies the beginning of a subroutine subprogram.

```
WRITE ( [UNIT=] unit [ [, write-spec ]... ] )
[ output-item [, output-item ]... ]
```

write-spec

is one of the following:

ERR = *lbl*	REC = *recno*
FMT= *format*	REPLY = *reply*
IOSTAT = *ios*	TIMEOUT = *to*
LENGTH = *len*	UNLOCK = *unlock*
MSGNUM = *msgno*	UPDATE = *ipd*

Outputs data to a specified unit.

Compiler Directives

This subsection specifies the syntax of all FORTRAN compiler directives.

```
[NO]ABORT
```

Specifies compiler action if it cannot open a file referenced in a SOURCE or CONSULT compiler directive.

Default is ABORT.

```
[NO]ANSI
```

Specifies that the compiler ignore characters beyond position 72 of a source line.

Default is NOANSI.

```
[NO]BOUNDSCHECK
```

Verifies that the subscripts in each reference to an element of array are within the bounds declared for the array.

Default is NOBOUNDSCHECK.

```
[NO]CODE
```

Lists the octal instruction codes generated for each program unit following the source listing for that unit.

Default is NOCODE.

```
COLUMNS number
```

Specifies that FORTRAN treat all text beyond the specified column as comments in each source line, beginning with the line that contains this directive.

Default is COLUMNS 132.

```
[NO]COMPACT
```

Specifies whether BINSERV should attempt to compact the code space of the target file.

Default is NOCOMPACT.

```
CONSULT { consult-item                          }
        { (consult-item [, consult-item]...)    }
```

consult-item

is

file-name [(*proc-name* [, *proc-name*]...)]

Declares procedures written in languages other than FORTRAN.

```
[NO]CROSSREF [identifier-class
             (identifier-class [, identifier-class]...)]
```

identifier-class

is one of the following:

BLOCKS	FUNCTIONS	PROGLABELS
BLOCKDATAS	GENERATE	STMTFUNCSs
CONSTANTS	INLINES	UNREF
DUMMYPROCS	LITERALS	VARIABLES
FMTLABELS	PROCEDURES	

Generates cross-reference information for selected identifier classes.

```
DATAPAGES number
```

Specifies the number of virtual memory pages to allocate for data storage.

```
ENDIF toggle
```

Terminates the effect of a preceding IF or IFNOT directive that specifies the same toggle number.

```
ENV {OLD   }
    {COMMON}
```

OLD specifies that the program use the C-series FORTRAN run-time library.

COMMON specifies that the program use the D-series FORTRAN run-time library. The D-series run-time library uses features of the Common Run-Time Environment (CRE).

```
ERRORFILE file-name
```

Saves compilation error messages in a disk file.

```
ERRORFILE number
```

Sets the maximum number of errors for a compilation.

Default is ERRORS 100.

```
[NO]EXTENDCOMMON
```

Instructs the compiler to use indexed indirect addressing to access simple variables in common blocks.

Default is NOEXTENDCOMMON.

```
[NO]EXTENDEDREF
```

Generates code that uses doubleword addresses for parameters in CALL statements and function references.

Default is NOEXTENDEDREF, unless you use the LARGECOMMON directive.

```
[NO]FIXUP
```

Instructs the compiler to omit some of the processing steps required to make an object file runnable.

Default is FIXUP.

```
[NO]FMAP
```

Includes a file map in the compiler's listing.

Default is NOFMAP.

```
GUARDIAN { procedure-name                                  }
         { (procedure-name [, procedure-name]...)          }
```

Declares procedures as Guardian procedures or as utility routines.

```
HIGHBUFFER size
```

If ENV OLD is in effect, specifies the number of words to allocate for the run-time buffer pool, #HIGHBUF, in upper data memory. If ENV COMMON is in effect, specifies the size of the CRE internal buffer area, #CRE_HEAP. By default, #CRE_HEAP is 1,024 words.

```
HIGHCOMMON [ block-name
           [ (block-name [, block-name ]...) ]
```

Allocates common storage in upper data memory for specified common blocks.

```
[NO]HIGHCONTROL
```

Allocates I/O control blocks in upper data memory.

Default is NOHIGHCONTROL.

```
[NO]HIGHPIN
```

Specifies that this FORTRAN program unit can run at a PIN that is greater than 255.

Default is NOHIGHPIN.

```
[NO]HIGHREQ
```

Specifies that Guardian processes that run at high PINs can open this FORTRAN program.

Default is NOHIGHREQ.

```
[NO]ICODE
```

Lists the symbolic instruction codes generated for each program unit following the source listing for that program unit.

Default is NOICODE.

```
IF toggle
```

If the specified toggle is set, the IF directive processes the source lines that follow, up to an ENDIF directive that specifies the same toggle number as the IF directive or to the end of the source file.

```
IFNOT toggle
```

If the specified toggle is reset, the IFNOT directive processes the following lines, up to an ENDIF directive that specifies the same toggle number as the IFNOT directive or to the end of the source file.

```
[NO]INSPECT
```

Establishes Inspect as the default debugger for the object file.

The Default is NOINSPECT.

```
{INTEGER*2}
{INTEGER*4}
{INTEGER*8}
```

Specifies the size of all subsequent entities in the source file declared explicitly or implicitly as INTEGER (without an explicit size specification).

```
LARGECOMMON {block-name                                  }
            {(block-name [, block-name]...)              }
```

Allocates space for the specified common blocks in extended memory.

```
LARGEDATA [item                  ]
          [(item [, item]...)    ]
```

Allocates memory space in the object program's extended data segment for local data.

```
LARGESTACK number
```

Specifies the block size to reserve for dynamically-allocated variables specified in LARGEDATA directives.

```
LIBRARY file-name
```

Establishes a default user library.

```
LINES number
```

Specifies the number of lines the compiler writes to the listing file before issuing a page skip.

```
[NO]LIST
```

Controls listing of source lines; enables the CODE, CROSSREF, ICODE, LMAP, MAP, and PAGE directives.

Default is LIST.

```
[NO]LMAP { list-option                          }
         { (list-option [, list-option]...)     }
```

Instructs BINSERV to pass load-map information to the compiler. The compiler lists the load maps after its identifier map and cross-reference tables.

Default is LMAP ALPHA.

```
{INTEGER*2}
{INTEGER*4}
```

Specifies the size of all subsequent entities in the source file that are declared as type logical.

```
LOWBUFFER size
```

Controls space allocated for the run-time buffer pool in lower data memory. If you specify ENV COMMON, FORTRAN recognizes the LOWBUFFER directive but does not allocate a #LOWBUFFER.

```
[NO]MAP
```

Lists, following each program unit's source listing, a table of local identifiers for that program unit. MAP also lists a table of entities in common storage following the last program unit's listing.

Default is MAP.

```
[NO]NONSTOP
```

Specifies that you want your program to run as a NonStop process pair.

Default is NONONSTOP.

```
PAGE [ " title" ]
```

Ejects the current page of the list file, prints the specified character string at the top of the next page, and skips two lines before resuming the listing.

```
POP { directive                          }
    { (directive [, directive]...)       }
```

Restores a directive to its original state from a push-down stack.

```
[NO]PRINTSYM
```

Includes or omits unreferenced identifiers in MAP listings.

Default is NOPRINTSYM.

```
PUSH { directive                          }
     { (directive [, directive]...)       }
```

Saves the current state of a compiler directive in a push-down stack.

```
RECEIVE { receive-spec                          }
        { (receive-spec [, receive-spec]...)    }
```

Specifies values for parameters that control the length of a reply, the number of processes that can open this process, the number of messages that can be posted to this process at any given time, the number of messages to be resent in the event of a failure, and whether you want to receive system messages.

```
RESETTOG [ toggle [, toggle ]... ]
```

Resets one or more specified toggles. If you do not specify toggle, RESETTOG resets all fifteen toggles. Toggles are specified by the numbers 1 through 15.

```
[NO]RUNNAMED
```

Specifies whether your program runs as a named process.

Default is NORUNNAMED.

```
SAVE { save-spec                          }
     { (save-spec [, save-spec]...)       }
```

save-spec

is one of the following:

```
STARTUP     PARAM     ASSIGNS     ALL
```

Specifies which messages—start-up message, param messages, or assign messages— you want FORTRAN to save so that your program can access them dynamically.

```
[NO]SAVEABEND
```

Specifies whether Inspect should automatically create a save file if the program terminates abnormally at run time.

Default is NOSAVEABEND.

```
SEARCH { file-name                              }
       { (file-name [, file-name]...)           }
```

Specifies a list of object files for BINSERV to search at compilation time for unsatisfied external references.

```
SECTION section-name
```

Assigns a name to a section of a source file for use in a SOURCE directive in another program.

```
SETTOG [ toggle [, toggle ]... ]
```

Sets one or more specified toggles for use as conditional compilation controls. If no toggle is specified, the compiler sets all fifteen toggles. Toggles are specified by the numbers 1 through 15.

```
SOURCE file-name [ ( section [, section ]... ) ]
```

Directs the compiler to read source lines from the specified file, either from the beginning of the file to the end of the file, or from the start of a specified section in the file to the end of the section.

```
SUBTYPE number
```

Specifies a process subtype for an object file.

Default is SUBTYPE 0.

```
[NO]SUPPRESS
```

Overrides the effect of the LIST directive. If SUPPRESS is active, the compiler lists only error messages and compilation statistics.

Default is NOSUPPRESS.

```
[NO]SYMBOLS
```

Specifies whether to include a symbol table in the object file for use by Inspect. You must specify SYMBOLS to use source-level debugging.

Default is NOSYMBOLS.

```
SYNTAX
```

Compiler scans source file for syntax errors but does not produce an object file.

```
UNIT { u[-u]                                              }
     { ( units [, file] [, create-spec]...)               }
```

Causes one or more units to exist and declares the properties of the file that will be connected to the unit.

```
[NO]WARN
```

Lists compiler warning messages, regardless of the setting of the LIST directive. NOWARN suppresses these messages.

Default is WARN.

C

Converting Programs to HP FORTRAN

This appendix contains suggestions to help you convert a FORTRAN application program that was not written for HP FORTRAN to the syntax and semantics of HP FORTRAN.

- Comment lines

  If the source program file has comments extending beyond column 72 of each line, add the directive line

  ```
  ? ANSI
  ```

  at the beginning of the file, so that HP FORTRAN will ignore all but the first 72 characters of each source line.

  Some FORTRAN implementations permit comments on the same source line as statements, but HP FORTRAN does not. Use an editor to move the comments to a separate comment line. Begin each comment line with a “C” or an asterisk in column 1.

- Storage allocation for integer, logical, and real data types

  If the program follows the ANSI FORTRAN 77 rules for storage allocation (which specify that variables of integer, logical, and real data types all occupy the same amount of storage space), add the directive line

  ```
  ? INTEGER*4, LOGICAL*4
  ```

  at the beginning of the source program file.

- Specifying variables for unit numbers

  If the program uses variables for unit numbers in I/O statements, so that the unit numbers do not “exist” as far as HP FORTRAN is concerned, add the directive line

  ```
  ? UNIT 1 - n
  ```

  at the beginning of the source program file, where n is the highest unit number used by the program. You might want to specify ranges of unit numbers, rather than specifying the entire range of unit numbers, because FORTRAN allocates space for a file control block for each unit that you specify in a unit directive.

- Mixing CHARACTER and other data types

  Neither the ANSI FORTRAN 77 standard nor HP FORTRAN support mixing variables of type CHARACTER with other data types in the same common block, but some other FORTRAN systems allow such mixing as an extension to the standard.

If you convert a program that uses a combination of CHARACTER and other data types in the same common block, you can declare a RECORD in the common block, and declare all the variables in the common block as components of the RECORD. (HP FORTRAN allows mixing of CHARACTER and other data types within a RECORD, since the entire RECORD feature is an HP extension.) If you do this, you must also change all references to those variables so that they are qualified by the RECORD name throughout all the executable statements of the source program.

- User and extended data segments

  In most computer systems, there is only one data area and all data addresses are the same length. NonStop systems have two data areas, the user data segment and the extended data segment, in which objects have 16-bit and 32-bit addresses, respectively. You should allocate smaller and more frequently used data objects in the user data segment for efficient access. Place larger and less frequently used objects in the extended data segment where more executable instructions are required to manipulate 32-bit addresses.

  If a FORTRAN program's data objects do not all fit into the user data segment (where HP FORTRAN places them by default), you can add the directive line

  ```
  ? LARGECOMMON, LARGEDATA
  ```

  at the beginning of the source program file. When the LARGECOMMON and LARGEDATA directives are in effect, the FORTRAN compiler allocates all COMMON blocks and all local data items larger than 256 bytes in the extended data segment.

  This is likely to make the recompiled object program much larger and somewhat slower, but it should at least run. Following the general principle "First make it run correctly, and then make it work better," you can determine which common blocks and local data items should be in which data area, based on their size and frequency of usage. Then modify the LARGECOMMON and LARGEDATA directives so the program makes optimal use of each data area.

- Allocation of local data objects

  Most FORTRAN systems support only static allocation of local data objects. That is, all variables and arrays are given run-time memory space when a program begins execution, and remain allocated throughout execution of the program.

  Because HP FORTRAN allows recursive procedure calls, local data objects are normally allocated space on the run-time stack at the time their procedure is entered, and their memory space is released when the procedure returns to its caller. HP FORTRAN provides static allocation for all variables in common blocks and for all local variables that are named in DATA statements or SAVE statements, but it provides dynamic allocation for all other local variables to conserve run-time memory space.

Dynamically allocated local variables do not retain their values between successive invocations of their procedure. This can cause failure of some programs that depend on such retention of local data values.

The ANSI FORTRAN 77 language includes a SAVE statement so that the programmer can specify explicitly which variables must be allocated statically, but many existing FORTRAN programs do not use the SAVE statement because it has no effect on a system that allocates all variables statically anyway.

If you suspect that a program you are converting might have been written with the assumption that all data is allocated statically, simply add a SAVE statement (with no variable list) to every subprogram. A SAVE statement with no variable list makes all variables in the subprogram static. You might want to be more selective to avoid permanently allocating data space.

- Redefining LOGICAL*1 or BYTE data types

  Some FORTRAN implementations include a LOGICAL*1 or BYTE data type, which is treated as unsigned 8-bit integer data, as an extension to the ANSI FORTRAN 77 standard. HP FORTRAN does not support this feature. You can usually replace references to LOGICAL*1 or BYTE data types with INTEGER*2, but you might have to add references to the CHAR and ICHAR intrinsic functions when combining data of types BYTE and CHARACTER.

- Redefining REAL*4 and REAL*8 data types

  Some FORTRAN implementations include the data type designators REAL*4 (synonymous with REAL) and REAL*8 (synonymous with DOUBLE PRECISION) as extensions to the ANSI FORTRAN 77 standard. To compile the program with HP FORTRAN, use an editor to replace all occurrences of REAL*4 with REAL and REAL*8 with DOUBLE PRECISION.

- Redefining COMPLEX*16 data types

  Some FORTRAN implementations include the data type COMPLEX*16 (meaning double-precision complex) as an extension to the ANSI FORTRAN 77 standard. You can use an editor to replace COMPLEX*16 with COMPLEX to make the program acceptable to HP FORTRAN, but of course this change will be accompanied by loss of precision in the results computed by the program.

- Expressions that exceed 255 characters

  The ANSI FORTRAN 77 standard does not specify the maximum length that a standard-conforming processor must support for variables, array elements, functions, constants, and expression values of type CHARACTER. Some FORTRAN systems support up to 32,767 characters but HP FORTRAN supports a maximum of 255 characters.

  If this limitation is a problem, you can try declaring the variable as a RECORD with components whose lengths total the required number of characters. HP FORTRAN allows you to use a RECORD name without a following circumflex (^) and

component name, almost anywhere that you can use a type CHARACTER variable.

- Special characters in symbolic names

  Some FORTRAN implementations allow additional characters such as “$” in symbolic names, as extensions to the ANSI FORTRAN 77 standard. You can usually use an editor to replace all such characters with blanks or an underscore character, to make the program acceptable to HP FORTRAN. Using an underscore character might help avoid converting a variable whose name includes a “$” to an already existing variable name that differs only in that it does not use a “$”.

- Record lengths

  HP FORTRAN requires that all records in a file be the same length, or (for some file types) be any length up to the maximum record length that is declared for the file. Some FORTRAN implementations allow unformatted READ and WRITE statements to have data lists of arbitrary length, with records of different lengths in the same file, and possibly with some records exceeding the maximum physical record length permitted by the host operating system. Programs that are dependent on such a “segmented records” feature might be difficult to convert to HP FORTRAN.

- Initializing data within a type declaration statement

  Some FORTRAN implementations allow initialization of data within type declaration statements. For example, the statement

```
REAL PI / 3.1415 9265 3589 7932 /
```

  would be equivalent to the two ANSI FORTRAN 77 statements

```
REAL PI
DATA PI / 3.1415 9265 3589 7932 /
```

  HP FORTRAN does not support this extension. You can use an editor to replace such a statement with two statements, as illustrated above.

- Initializing variables in common blocks

  The ANSI FORTRAN 77 standard and HP FORTRAN require that DATA statements that initialize variables in common blocks appear only in BLOCK DATA program units. Some FORTRAN implementations allow you to initialize data in common blocks in any executable program unit as well. You must move all such DATA statements into a separate BLOCK DATA program unit.

- Octal and hexadecimal constants in source code

  The ANSI FORTRAN 77 standard makes no provision for octal or hexadecimal constants in source code. HP FORTRAN supports octal constants written as %*nnnnnn* as an extension to the standard. Other FORTRAN implementations also allow octal and hexadecimal constants, but use a different notation from that of HP

FORTRAN. Use an editor to find all such occurrences and replace them with forms that are acceptable to HP FORTRAN.

Some FORTRAN implementations also provide octal and hexadecimal conversions in formatted I/O, again with a variety of syntaxes. HP FORTRAN provides octal and hexadecimal conversions, but the HP-defined syntax might vary from the syntax used in your program. Use an editor to find all such occurrences and replace them with forms that are acceptable to HP FORTRAN.

- Converting ENCODE and DECODE statements

  Some FORTRAN implementations support the ENCODE and DECODE statements as extensions to the ANSI FORTRAN 66 standard. These were replaced in the ANSI FORTRAN 77 standard by the “internal file” feature, that is, the use of a type CHARACTER variable in place of the unit number in a READ or WRITE statement. You can do likewise if you encounter any ENCODE or DECODE statements in programs you are converting.

- Converting NAMELIST I/O

  Some FORTRAN implementations support NAMELIST I/O as an extension to the ANSI FORTRAN 66 standard. NAMELIST I/O was omitted from the ANSI FORTRAN 77 standard because it was used so infrequently. NAMELIST I/O is not included in HP FORTRAN. Programs that use this feature can be difficult to convert to HP FORTRAN.

- Intrinsic functions and logical operators

  Some FORTRAN implementations support the intrinsic functions AND, OR, XOR, and COMPL, and/or the use of logical operators .AND., .OR., .XOR., and .NOT., to perform bitwise masking operations on INTEGER or REAL values, as extensions to the ANSI FORTRAN 66 standard. Neither ANSI FORTRAN 77 standard nor HP FORTRAN support this capability. You can write TAL subprograms for the AND, OR, XOR, and COMPL functions.

- Direct-access READ and WRITE statements

  Some FORTRAN implementations support direct-access READ and WRITE statements of the form

  ```
  READ (u ' rn [ , ERR = label ] ) datalist
  WRITE (u ' rn [ , ERR = label ] ) datalist
  ```

  where u is an I/O unit number and

  rn is a record number,

  as extensions to the ANSI FORTRAN 66 language. These were replaced in ANSI FORTRAN 77 by the statement forms

  ```
  READ (u, REC = rn [ , ERR = label ] ) datalist
  WRITE (u, REC = rn [ , ERR = label ] ) datalist
  ```

HP FORTRAN supports the ANSI FORTRAN 77 form but not the obsolete extension. If you encounter such statements in programs you are converting, change them as shown above.

- Converting programs that read blocked tapes

  If a program you are converting reads blocked tapes, you can either cause the tapes to be copied to an unblocked file for use by a HP FORTRAN program, or change the FORTRAN program to do its own unblocking of the tape records.

- Hollerith constant syntax

  Some FORTRAN implementations allow Hollerith constants with single and double quotation mark characters (' and ") as delimiters, as well as the nHcc...c notation specified in an appendix to the ANSI FORTRAN 77 standard. HP FORTRAN follows that appendix.

  HP FORTRAN diagnoses all uses of the " character, but the ' character looks like a character (not Hollerith) constant, so you will get an error message only if such a constant is used in a way that is not allowed for character constants. Replace all such Hollerith constants with the ANSI notation.

- Using Hollerith constants

  Some FORTRAN implementations allow Hollerith constants anywhere that integer, real, and double precision constants are allowed. HP FORTRAN follows the appendix to the ANSI FORTRAN 77 standard that restricts Hollerith constants to DATA and CALL statements.

  Programs that use Hollerith constants outside of DATA and CALL statements might be difficult to convert to HP FORTRAN. You must find all the Hollerith constants not used in DATA and CALL statements, change them to character constants, and change the declarations of all non-character variables used with them to type character of the appropriate length. This change, however, might cause syntax errors because other non-character variables are also used with those variables. You might have to make many passes through the program, changing more variables to type character, until you can compile your program without syntax errors.

D

Data Type Correspondence and Return Value Sizes

The following tables contain the return value size generated by HP language compilers for each data type. Use this information when you need to specify values with the Accelerator ReturnValSize option. These tables are also useful if your programs use data from files created by programs in another language, or your programs pass parameters to programs written in callable languages.

Refer to the appropriate NonStop SQL programmer's guide for a complete list of SQL data type correspondence. Also note that the return value sizes given in these tables do not correspond to the storage size of SQL data types.

Note. COBOL includes COBOL 74, COBOL85, and SCREEN COBOL unless otherwise noted.

If you are using the Data Definition Language (DDL) utility to describe your files, you might not need this table. For more information, refer to the *Data Definition Language (DDL) Reference Manual*.

Table D-1. Integer Types, Part 1 (page 1 of 2)

	8-Bit Integer	16-Bit Integer	32-Bit Integer
BASIC	STRING	INT INT(16)	INT(32)
C	char [1] unsigned char signed char	int short unsigned	long unsigned long
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 1≤n≤4 Index Data Item [2] NATIVE-2 [3]	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 5≤n≤9 Index Data Item [2] NATIVE-4 [3]
FORTRAN	—	INTEGER [4] INTEGER*2	INTEGER*4
Pascal	BYTE Enumeration, unpacked, ≤256 members Subrange, unpacked, n…m, 0 ≤n and m≤255	INTEGER INT16 CARDINAL [1] BYTE or CHAR value parameter Enumeration, unpacked, > 256 members Subrange, unpacked, n…m, -32768 ≤n and m≤ 32767, but at least n or m outside 0…255 range	LONGINT INT32 Subrange, unpacked n…m, –2147483648 £ n and m ≤2147483647, but at least n or m outside -32768…32767 range

Table D-1. Integer Types, Part 1 (page 2 of 2)

	8-Bit Integer	16-Bit Integer	32-Bit Integer
SQL	CHAR	NUMERIC(1)…NUMERIC(4) PIC 9(1) COMP…PIC 9(4) COMP SMALLINT	NUMERIC(5)…NUMERIC(9) PIC 9(1) COMP…PIC 9(9) COMP INTEGER
TAL	STRING UNSIGNED(8)	INT UNSIGNED(16)	INT(32)
Return Value Size (Words)	*1*	*1*	*2*

[1] Unsigned Integer.

[2] Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in COBOL85.

[3] COBOL85 only.

[4] INTEGER is normally equivalent to INTEGER*2. The INTEGER*4 and INTEGER*8 compiler directives redefine INTEGER.

Table D-2. Integer Types, Part 2

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer
BASIC	INT(64) FIXED(0)	—	—
C	long long	—	—
COBOL	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 10 ≤n ≤18 NATIVE-8 [1]	—	Numeric DISPLAY
FORTRAN	INTEGER*8	—	—
Pascal	INT64	UNSIGNED(n), 1 ≤n ≤ 16 INT(n), 1≤n ≤16	DECIMAL
SQL	NUMERIC(10)…NUMERIC(18) PIC 9(10) COMP…PIC 9(18) COMP INTEGER	—	DECIMAL (n,s) PIC 9(n) DISPLAY
TAL	FIXED(0)	UNSIGNED(n), 1 ≤n ≤ 31	—
Return Value Size (Words)	*4*	*1, 1 or 2 in TAL*	*1 or 2, depends on declared pointer size*

[1] COBOL85 only.

Table D-3. Floating, Fixed, and Complex Types

	32-Bit Floating	64-Bit Floating	64-Bit Fixed Point	64-Bit Complex
BASIC	REAL	REAL(64)	FIXED(s), 0 ≤ s ≤18	—
C	float	double	—	—
COBOL	—	—	PIC S9(n–s)v9(s) COMP or PIC 9(n–s)v9(s) COMP, 10 ≤n ≤18	—
FORTRAN	REAL	DOUBLE PRECISION	—	COMPLEX
Pascal	REAL	LONGREAL	—	—
SQL	—	—	NUMERIC (n,s) PIC 9(n-s)v9(s) COMP	—
TAL	REAL	REAL(64)	FIXED(s), -19 ≤ s ≤1	—
Return Value Size (Words)	*2*	*4*	*4*	*4*

Table D-4. Character Types (page 1 of 2)

	Character	Character String	Varying Length Character String
BASIC	STRING	STRING	—

C	signed char
	unsigned char
		pointer to char	struct {
			int len;
			char val [n]
			};

COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	01 name. 03 len USAGE IS NATIVE-2 [1] 03 val PIC X(n).
FORTRAN	CHARACTER	CHARACTER array CHARACTER*n	—
Pascal	CHAR or BYTE value parameter Enumeration, unpacked, ≤256 members Subrange, unpacked n…m, 0 ≤n and m ≤255	PACKED ARRAY OF CHAR FSTRING(n)	STRING(n)

Table D-4. Character Types (page 2 of 2)

	Character	Character String	Varying Length Character String
SQL	PIC X CHAR	CHAR(n) PIC X(n)	ARCHAR(n)
TAL	STRING	STRING array	—
Return Value Size (Words)	*1*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*

[1] COBOL85 only.

Table D-5. Structured, Logical, Set, and File Types

	Byte-Addressed	Structure Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
BASIC	—	MAP buffer	—	—	—	—
C	—	struct	—	—	—	—
COBOL	—	01-level RECORD	—	—	—	—
FORTRAN	RECORD	—	LOGICAL [1]	—	—	—
Pascal	RECORD, byte-aligned	RECORD, word-aligned	—	BOOLEAN	Set	File
SQL	—	—	—	—	—	—
TAL	Byte-addressed standard STRUCT pointer	Word-addressed standard STRUCT pointer	—	—	—	—
Return Value Size (Words)	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on compiler directive*	*1*	*1*	*1*

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.

Table D-6. Pointer Types

	Procedure Pointer	Byte Pointer	Word Pointer	Extended Pointer
BASIC	—	—	—	—
C	function pointer	byte pointer	word pointer	extended pointer
COBOL	—	—	—	—
FORTRAN	—	—	—	—
Pascal	Procedure pointer	Pointer, byte-addressed BYTEADDR	Pointer, byte-addressed WORDADDR	Pointer, extended-addressed EXTADDR
SQL	—	—	—	—
TAL	—	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
Return Value Size (Words)	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*

E Compiler Limits

This appendix summarizes the limits of the FORTRAN compiler.

- Symbolic names can be up to 31 characters long. FORTRAN discards characters beyond the 31st. Therefore, the following two identifiers are the same:

  ```
  abcdefghijklmnopqrstuvwxyz123456
  abcdefghijklmnopqrstuvwxyz123457
  ```

- Executable programs

  An executable program must have exactly one main program.

  The object code for an executable program can have up to 16 memory segments (65,536 words per segment) if you use a user library object file in addition to the program file. If you do not use a user library, your program file can have up to 32 segments.

- Statements

  A source statement can have one initial line and up to 19 continuation lines, or a total of 20 lines per statement.

  A source statement can be composed of up to 1,320 characters. (A statement using columns 7 through 72 of each of 20 lines uses 1,320 characters.) A source statement can contain more than 1,320 characters if you use columns beyond column 72 to contain program text.

A computed GO TO statement can specify to 255 statement labels in its list.

- Subprograms

  The object code for a single program unit can be up to 32,768 words, including all executable instructions and read-only data such as constants and translated FORMAT statements, but not including local variables and arrays.

  A single program unit can contain up to 501 ASSIGN statements.

  External subprograms can have up to 63 dummy arguments.

  A subprogram that includes ENTRY statements can have up to 63 unique dummy arguments; a dummy argument for more than one entry point in the same subprogram is counted only once.

- Statement functions

  A statement function can have any number of dummy arguments as long as the dummy arguments do not require more than 31 words of storage. The space required for each type of parameter is:

INTEGER*2	1 word
INTEGER*4	2 words
INTEGER*8	4 words
LOGICAL mode	1 word, or 2 words if LOGICAL*4
REAL	2 words
DOUBLE PRECISION	4 words
COMPLEX	4 words
CHARACTER	1 word, or 2 words if EXTENDEDREF mode, regardless of the number of characters

- Data types

  Character variables, array elements, functions, constants, and expression values can be up to 255 characters in length.

  The total size of a RECORD, or of each element of a RECORD array, can be up to 32,767 bytes.

  RECORD declarations can be nested up to 15 deep, including the outermost RECORD and the fields within the innermost RECORD.

  A character expression that consists of a series of concatenated values can have up to 64 such values.

- Arrays

  A non-RECORD array can have up to seven dimensions. A RECORD, or an array within a RECORD, can have at most one dimension.

  The subscript bounds for each dimension of an array in the user data segment must be in the range -32,768 through 32,767, and the total size of the array can be up to 65,536 bytes. An array in the extended data segment can have subscript bounds in the range -2,147,483,648 through 2,147,483,647, and a total size of up to 133,693,140 bytes.

- Common blocks

  A common block can contain either exactly one RECORD and no other variables, or any number of non-RECORD variables and arrays.

  The total number of common blocks an executable program can have is unlimited, but no one program unit can declare more than 64 common blocks.

- Units and files

I/O unit numbers must be in the range 1 through 999. You can declare up to 128 different unit numbers in a compilation.

Data files can have record lengths up to 32,767 bytes, but for most file types the limit is 4,096 bytes. Consult the *ENSCRIBE Programmer's Guide* for details.

- Records in source files and listing files can be up to 132 characters long.

  Source files referenced by a SOURCE compiler directive can include further SOURCE compiler directives, up to a maximum nesting depth of six levels.

  The compiler has other limits such as the symbol table size, the amount of data it can store from DATA and FORMAT statements, and the number of external procedures that you can define and reference in an entire compilation. These limits cannot be expressed in a simple way because the internal table entries involved are variable in length. For example, the size of a symbol table entry depends on several factors including the length of the symbolic name and the number of dimensions.

  If the compiler issues an error message saying you have exceeded a compiler table size limit, divide your source file into two or more smaller ones with fewer program units in each, and then use separate compiler runs to compile the whole program.

Compile-Time Diagnostic Messages

This appendix lists the FORTRAN 77 compiler diagnostic messages that FORTRAN might report in the program listing. The compiler lists a diagnostic message immediately after it detects the condition displayed; however, because an error might be the result of an internal condition or an improper relationship between statements, the message might not appear immediately following the statement that caused the condition.

Compiler diagnostic messages are of two types: error messages, which indicate conditions that are serious enough to prevent the creation of an object program, and warning messages, which indicate conditions of less severity. The forms of the compiler diagnostic messages are:

```
**** ERROR ***** message-text
```

and

```
**** WARNING ***** message text
```

FORTRAN tries to complete compilation after issuing a warning message but the compiler must make assumptions about the situation that caused the warning in order to continue. As a result, the compiled program might not be what you intended, even if the compilation completes.

The entry for each warning message explains what FORTRAN does about the situation that caused the warning. If you use a program from a compilation that included a warning message, check the entry for the warning message to make sure that the compiler's action corresponded to what you intended. In most cases, you'll need to correct the program and recompile.

Topics covered in this section include:

The compiler's error messages and warning messages are presented in alphabetical order, with an explanation of the condition that caused FORTRAN to report each message. For an error message, the recommended procedure to correct the error condition is also given.

Error Messages

```
A CONSTANT IS EXPECTED HERE
```

Cause. The PARAMETER statement requires a constant on the right of the assignment operator.

Recovery. Correct the statement.

```
ADJUSTABLE ARRAY MUST BE DUMMY ** array-name
```

Cause. The upper or lower bound of an array that is not a dummy argument is a variable.

Recovery. Change the adjustable dimension specification to a constant, or make the array a dummy argument.

```
ADJUSTABLE LENGTH MUST BE FORMAL ** name
```

Cause. A type character variable or array that is not a dummy argument has a length specification that is a variable.

Recovery. Change the adjustable length specification to a constant, or make the character variable or array a dummy argument.

```
ARITHMETIC OVERFLOW
```

Cause. Evaluation of a constant expression at compilation time resulted in an arithmetic overflow.

Recovery. Correct or rearrange the constant expression.

```
ARRAY FORCED TO PRECEDE COMMON AREA ** array-name
```

Cause. An EQUIVALENCE specification has forced an entity to precede the common block in which it resides.

Recovery. Correct the EQUIVALENCE specification.

```
ARRAY HAS MORE THAN 32767 ELEMENTS ** name
```

Cause. The EXTENDEDREF compiler directive is not specified or implied, and the total number of elements in the indicated array is too large to be indexed by an INTEGER*2 value.

Recovery. Reduce the array size or use a LARGECOMMON or LARGEDATA compiler directive so that FORTRAN allocates the array in the extended data segment.

```
ARRAY INDEX OUT OF BOUNDS IN DATA STATEMENT ** name
```

Cause. A DATA statement attempts to initialize a nonexistent element of an array.

Recovery. Correct the DATA statement.

```
ARRAY SIZE EXCEEDS 65535 BYTES ** name
```

Cause. An array is too large to fit into either half of the user data segment.

Recovery. Reduce the array size or use a LARGECOMMON or LARGEDATA compiler directive so that FORTRAN allocates the array in the extended data segment.

```
ASSUMED-SIZE ARRAY MUST BE DUMMY ** array-name
```

Cause. An array that is not a dummy argument has an asterisk as the upper bound of its last dimension.

Recovery. Change the assumable dimension specification to a constant or make the array a dummy argument.

```
BLOCK DATA CANNOT INITIALIZE NON-COMMON DATA ** name
```

Cause. A DATA statement within a block data program unit initializes a variable or array that is not in any common block.

Recovery. Correct the DATA statement (the variable name might be misspelled) or use a COMMON or EQUIVALENCE statement to add the variable or array to a common block.

```
CHARACTER CONSTANT TOO LARGE
```

Cause. A character constant or Hollerith constant exceeds 255 characters in length.

Recovery. Shorten the constant.

```
CHARACTER ENTRY MIXED WITH OTHER TYPES
```

Cause. A function subprogram includes one or more ENTRY statements and the function name and the entry names are not all character type or are not all noncharacter types.

Recovery. Declare all entries character type or all entries noncharacter types.

```
CHARACTER ITEM IN HIGH COMMON ** block-name
```

Cause. A HIGHCOMMON compiler directive has forced a common block that contains character data into the upper half of the user data segment. The upper half of the user data segment is not byte addressable.

Recovery. Correct the directive.

```
CHARACTER TYPE MIXING NOT ALLOWED IN COMMON ** name
```

Cause. The indicated common block contains character type and noncharacter type entities. You can mix character type data and noncharacter type data only in a RECORD.

Recovery. Segregate entities of character and noncharacter data types into different common blocks or make the entire common block a RECORD. If you choose the latter, you must also change all references to those entities throughout the program, so that they are qualified by the name of the RECORD.

```
CHARACTER TYPE MIXING NOT ALLOWED IN EQUIVALENCE
```

Cause. An EQUIVALENCE statement names an entity of character type and an entity of a noncharacter data type in the same equivalence class. This is permitted only when the entities are within a RECORD.

Recovery. Correct the EQUIVALENCE statement.

```
CODE SPACE OVERFLOW
```

Cause. The program unit requires more than 32,767 words for object code, including executable instructions and read-only data such as constants and FORMAT statements. The compiler stops immediately after issuing this error message.

Recovery. Reduce the size of the program unit.

```
COMMON BLOCK SIZE EXCEEDS 65535 BYTES ** block-name
```

Cause. The indicated common block is too large for either half of the user data segment.

Recovery. Reduce the total size of the common block or use the LARGECOMMON compiler directive to allocate the common block in the extended data segment.

```
COMPILER TABLE OVERFLOW ** table-name
```

Cause. One of the compiler's internal tables is full. The compiler stops immediately after issuing this error message.

Recovery. Notify the Global Customer Support Center (GCSC).

```
CONTRADICTORY DEFAULT COMMON ALLOCATION
```

Cause. A HIGHCOMMON compiler directive and a LARGECOMMON directive were specified without a common block name on either directive.

Recovery. Add block names to one of the directives or delete one of the directives.

```
CONTROL EXPRESSION MUST BE ARITHMETIC
```

Cause. The control expression of an arithmetic IF statement is not type INTEGER, REAL, or DOUBLE PRECISION.

Recovery. Correct the expression.

```
CONTROL EXPRESSION MUST BE LOGICAL
```

Cause. The control expression of a logical IF, block IF, or ELSE IF statement is not type LOGICAL.

Recovery. Correct the expression.

```
CONTROL STATEMENT OUT OF ORDER
```

Cause. A block IF statement sequence incorrectly overlaps a DO loop body or another block IF statement sequence.

Recovery. Correct the statement sequence.

```
CPU TNS CONFLICTS WITH EXTENDEDREF
```

Cause. The directives specify that the object program is to be executable on NonStop 1+ systems and is to use extended addressing, which is not allowed on NonStop 1+ systems.

Recovery. Either remove the CPU TNS directive, or remove all EXTENDEDREF, LARGECOMMON, and LARGEDATA directives.

```
DATA ITEM IS NOT A RECORD ITEM
```

Cause. Reference has been made to a nonexistent RECORD field of the form record-name^ field-name.

Recovery. Correct the reference or the RECORD declaration.

```
DATA STATEMENT WITH COMMON ALLOWED ONLY IN BLOCK DATA **
variable-or-array-name
```

Cause. A DATA statement, not within a block data program unit, is trying to initialize a variable or array that is in a common block.

Recovery. Place the DATA statement in a block data subprogram.

```
DIRECTIVE MUST APPEAR BEFORE FORTRAN STATEMENTS ** dir
```

Cause. The *dir* compiler directive must precede the first FORTRAN language statement in the compilation. FORTRAN ignores the directive.

Recovery. Move the directive.

```
DIRECTIVE MUST APPEAR BETWEEN PROGRAM UNITS ** dir
```

Cause. The dir compiler directive must precede the first FORTRAN language statement in a program unit. FORTRAN ignores the directive.

Recovery. Move the directive.

```
DIRECTIVE REQUIRES EXTENDEDREF BEFORE ANY STATEMENTS ** dir
```

Cause. The dir compiler directive must be preceded by an EXTENDEDREF, LARGECOMMON, or LARGEDATA directive and must appear before the first FORTRAN statement in a program unit. The compiler ignores the directive.

Recovery. Move the directive.

```
DUMMY ARGUMENT IN COMMON ** name
```

Cause. The same name appears in both a dummy argument list and a COMMON statement in the same subprogram.

Recovery. Eliminate one occurrence.

```
DUMMY ARGUMENT PREVIOUSLY REFERENCED IN EXECUTABLE ** name
```

Cause. This dummy argument of an ENTRY has been referenced previously in an executable statement or a statement function, but did not appear in a preceding ENTRY, FUNCTION, or SUBROUTINE statement.

Recovery. Correct the ENTRY statement or the preceding code.

```
DUMMY NAME CONFLICT ** name
```

Cause. This dummy argument name is the same as a procedure name appearing in a FUNCTION, SUBROUTINE, or ENTRY statement in the same program unit.

Recovery. Change the dummy argument name.

```
DUPLICATE DUMMY VARIABLE ** name
```

Cause. This symbolic name appears more than once in a dummy argument list in the same FUNCTION, SUBROUTINE, or ENTRY statement.

Recovery. Correct the list.

```
DUPLICATE IMPLICIT ENTRY
```

Cause. The same letter appears in two IMPLICIT statements.

Recovery. Change one of the statements.

```
DUPLICATE STATEMENT LABEL
```

Cause. Two statements within the same program unit have the same statement label.

Recovery. Make sure all statement labels are unique.

```
END STATEMENT MISSING
```

Cause. The source input file ended with no END statement to terminate the last (or only) source program unit.

Recovery. Correct the source input file.

```
ENTRY NESTED IN CONTROL LOOP
```

Cause. An ENTRY statement appears within the body of a DO loop or a block IF statement sequence.

Recovery. Correct the statement sequence.

```
ERROR IN FORMAT SPECIFICATION
```

Cause. A FORMAT specification contains a syntax error.

Recovery. Correct the format specification.

```
ERROR IN NUMERIC CONSTANT
```

Cause. A numeric constant contains a syntax error.

Recovery. Correct the constant.

```
ERROR IN RECORD EQUIVALENCE
```

Cause. The program attempts to equivalence RECORDs that are declared in different common blocks.

Recovery. Either remove the EQUIVALENCE statement, or else delete one of the RECORD names from the COMMON statement in which it is declared.

```
EXPECTED A FORMAT
```

Cause. A FORMAT specification is expected in this context.

Recovery. Correct the statement.

```
EXPECTS DUMMY VARIABLE
```

Cause. A dummy argument is expected in this context.

Recovery. Correct the statement.

```
EXTENDED DATA ADDRESSES INVALID FOR GUARDIAN CALLS **
variable-or-array-name
```

Cause. You specified a data item with a doubleword address as a pass-by-reference argument to a Guardian procedure that expects a word address. A variable or array has a doubleword address if it is a formal parameter in a subprogram compiled with the EXTENDEDREF compiler directive specified or implied, or if it is allocated in the extended data segment as a result of a LARGECOMMON or LARGEDATA compiler directive.

Recovery. Use the D-series version of the Guardian routine if one exists. Otherwise, use a local variable or array as an argument in the procedure call. Assign the value of the extended-address data item to the local data item before the procedure call, and assign the value of the local data item to the extended-address variable or array after the procedure call.

```
EXTRANEOUS CHARACTERS IN CONSTANT
```

Cause. A constant contains unexpected characters.

Recovery. Correct the constant.

```
FILLERS CAN ONLY BE WITHIN RECORDS
```

Cause. A FILLER specification occurs in a context other than a RECORD declaration.

Recovery. Correct the source.

```
FORMAL PARAMETERS MAY NOT BE EQUIVALENCED ** name
```

Cause. A dummy argument appears in an EQUIVALENCE statement.

Recovery. Correct the statement.

```
FUNCTION MUST BE ALPHABETIC
```

Cause. A function name begins with a character other than a letter of the alphabet.

Recovery. Correct the function name.

```
FUNCTION TYPE INCONSISTENT
```

Cause. A function has been defined as one data type and referred to as another.

Recovery. Make the definition and references consistent.

```
FUNCTIONS MAY NOT BE EQUIVALENCED ** name
```

Cause. A function name appears in an EQUIVALENCE statement. Possibly it was intended to be an array name, but the array declaration is missing or in error. FORTRAN assumes any name followed by "(" is a function if the name has not been declared as an array name.

Recovery. Correct the statement, or add the array declaration.

```
GENERIC FUNCTION NOT DEFINED FOR ARGUMENT TYPE ** name
```

Cause. You referenced a generic intrinsic function with an argument whose data type is not one of the types for which the generic function is defined. Example: ABS (X) where X is a type character variable.

Recovery. Correct the source statement.

```
IDENTIFIER ALREADY DECLARED
```

Cause. A symbolic name is illegally declared twice.

Recovery. Remove one declaration.

```
IDENTIFIER ALREADY TYPED
```

Cause. A symbolic name appears in more than one data type declaration statement.

Recovery. Remove one data type declaration.

```
ILLEGAL ARITHMETIC OPERAND
```

Cause. An operand of some other data type appears in a context that requires an integer, real, double precision, or complex value.

Recovery. Correct the expression.

```
ILLEGAL ARITHMETIC OPERATOR
```

Cause. A logical or character operator appears in a context that requires an arithmetic operator.

Recovery. Correct the expression.

```
ILLEGAL CHARACTER LENGTH
```

Cause. The length of a CHARACTER datum is not in the range 1 through 255, or an adjustable-length CHARACTER datum appears in a concatenation that is not in an assignment statement.

Recovery. Correct the statement.

```
ILLEGAL CHARACTER OPERAND
```

Cause. An operand of some other data type appears in a context that requires a type character value.

Recovery. Correct the expression.

```
ILLEGAL CHARACTER OPERATOR
```

Cause. An arithmetic or logical operator appears in a context that requires a character operator.

Recovery. Correct the expression.

```
ILLEGAL COMBINATION OF ATTRIBUTES
```

Cause. The attributes specified for an identifier are incompatible with each other.

Recovery. Correct the specifications.

```
ILLEGAL COMBINATION OF SPECIFIERS
```

Cause. The control list of an INQUIRE statement includes both UNIT and FILE specifiers. Or, the control list of a POSITION statement includes a REC specifier and also any of KEY, KEYLEN, KEYID, MODE, COMPARELEN, and SKIPEXACT specifiers.

Recovery. Correct the statement.

```
ILLEGAL LAST STATEMENT FOR DO
```

Cause. The body of a DO loop cannot end with this type of statement.

Recovery. Use a CONTINUE statement or otherwise correct the DO loop.

```
ILLEGAL LOGICAL OPERAND
```

Cause. An operand of some other data type appears in a context that requires a type logical value.

Recovery. Correct the expression.

```
ILLEGAL LOGICAL OPERATOR
```

Cause. An arithmetic or character operator appears in a context that requires a logical operator.

Recovery. Correct the expression.

```
ILLEGAL TYPE CONVERSION
```

Cause. The data type of a constant is not compatible with the data type required and the necessary conversion is not possible.

Recovery. Correct the statement.

```
ILLEGAL TYPE MIXING
```

Cause. An expression contains incompatible operands.

Recovery. Correct the expression.

```
ILLEGAL USE OF ASSUMED SIZE ARRAY ** array-name
```

Cause. The name of a dummy array that was declared with an asterisk as the upper bound of its last dimension, appears without subscripts as the unit, the format, or a data list item in an I/O statement.

Recovery. Replace with another array or change the statement.

```
ILLEGAL USE OF EXTERNAL
```

Cause. An external function name is used as a variable.

Recovery. Use the identifier as a function name or a variable name, but not both.

```
ILLEGAL USE OF FORMAT
```

Cause. A FORMAT statement label appears where the label of an executable statement is required.

Recovery. Supply the proper label.

```
IMPLICIT ITEMS MUST BE ONE LETTER
```

Cause. An IMPLICIT statement entry must be a single letter or a range of single letters.

Recovery. Correct the statement.

```
IMPLICIT STATEMENT RETYPES PARAMETER ** name
```

Cause. An IMPLICIT statement would change the data type of a symbolic constant that was previously declared in a PARAMETER statement.

Recovery. Declare the symbolic constant's data type explicitly.

```
IMPROPER EQUIVALENCE IN A RECORD
```

Cause. An EQUIVALENCE statement within a RECORD declaration does not equivalence components at its own level.

Recovery. Correct the statement.

```
INCORRECT FORM FOR ELSE IF STATEMENT
```

Cause. An ELSE IF statement contains a syntax error.

Recovery. Correct the statement.

```
INSUFFICIENT ARRAY SIZE FOR LENGTH
```

Cause. In an I/O statement that has a character array as an internal file (in place of the unit designator), the LENGTH specifier (an HP extension) does not name an array of at least the same number of elements.

Recovery. Change the array's declaration or make the LENGTH specifier designate a different array.

```
INSUFFICIENT NUMBER OF DATA CONSTANTS
```

Cause. A constant list in a DATA statement has fewer items than the corresponding name list.

Recovery. Supply the missing constants.

```
INSUFFICIENT TEXT FOR HOLLERITH CONSTANT
```

Cause. A Hollerith character string contains fewer characters than were specified.

Recovery. Make the statement long enough to have the indicated number of characters following the H.

```
INTEGER OVERFLOW
```

Cause. An integer constant in a compiler directive that requires a value in the range -32768 through +32767 is outside this range.

Recovery. Correct the constant.

```
INVALID BOUNDS FOR SUBSTRING
```

Cause. The upper bound of a substring is less than its lower bound, or a bound is outside the range 1 through 255, or the upper bound exceeds the length of the variable or array element.

Recovery. Correct the substring specification.

```
INVALID DATA ITEM ** name
```

Cause. The name list of a DATA statement includes something other than a variable name, an array name, an array element, or a substring. In particular, RECORDs, RECORD fields, and dummy arguments are not allowed.

Recovery. Correct the statement.

```
INVALID FILE NAME
```

Cause. The file name in a UNIT compiler directive contains a syntax error.

Recovery. Correct the file name.

```
INVALID FORM FOR A FILLER
```

Cause. A FILLER specification contains a syntax error.

Recovery. Correct the specification.

```
INVALID FORTRAN CHARACTER
```

Cause. A character was found that is not in the FORTRAN character set.

Recovery. Remove or replace the character.

```
INVALID I/O CONTROL ** keyword
```

Cause. The indicated control specifier in an I/O statement contains a syntax error.

Recovery. Correct the statement.

```
INVALID IMPLICIT RANGE
```

Cause. A range specification in an IMPLICIT statement is not in alphabetical order.

Recovery. Reverse the range specification.

```
INVALID IMPLICIT STATEMENT
```

Cause. A IMPLICIT statement contains a syntax error.

Recovery. Correct the statement.

```
INVALID KEY WORD
```

Cause. The beginning of a statement is not recognizable as a valid FORTRAN statement.

Recovery. Correct the statement.

```
INVALID STATEMENT NUMBER ** label
```

Cause. The source statement references a statement label that is more than five digits.

Recovery. Correct the source statement. This problem can arise when the source program has line sequence numbers to the right of column 72, which are treated as part of the statement (remember that FORTRAN ignores blanks). In this case, use the ANSI compiler directive so that the compiler scans only columns 1 through 72.

```
INVALID TYPE FOR THIS CONTEXT
```

Cause. A constant or expression of a non-integer data type appears in an array declarator as a dimension's upper or lower bound, or as a bound in a substring name, or as a repetition count in the constant list of a DATA statement, or as a unit number in an I/O statement. Or, a constant or expression of a non-character data type appears as a format in an I/O statement.

Recovery. Correct the statement.

```
INVALID TYPE FOR THIS CONTEXT ** keyword
```

Cause. The indicated control specifier in an I/O statement designates an entity of an unacceptable data type.

Recovery. Correct the statement.

```
INVALID TYPE FOR THIS CONTEXT ** name
```

Cause. The indicated variable or array is used as a format designator in an I/O statement but its data type is not character.

Recovery. Correct the statement.

```
INVALID TYPE STATEMENT
```

Cause. A data type statement contains a syntax error.

Recovery. Correct the statement.

```
INVALID USE OF ASTERISK
```

Cause. An assignment statement or statement function definition begins with a phrase that contains an asterisk such as INTEGER*2.

Recovery. Correct the statement.

```
ITEM MAY NOT BE BY VALUE ** keyword
```

Cause. An I/O statement control specifier in which a result is to be stored, is a constant or expression other than a variable or an array element. Or an internal file (a character entity appearing in place of an I/O unit number) is a constant, an expression other than a variable, or an array or array element.

Recovery. Correct the statement.

```
LOCAL AREA OVERFLOW
```

Cause. This program unit declares more than 32,767 words of local variables, arrays, and RECORDs in the standard data segment.

Recovery. Reduce the number or size of such local variables, by doing one or more of the following:

- Make array dimensions smaller.
- Move some local variables, arrays, and RECORDs to one or more new common blocks.
- Move some local variables, arrays, and RECORDs to the extended data segment with the LARGEDATA compiler directive.

```
LOGIC ERROR IN COMPILER
```

Cause. The compiler is in an inconsistent state as a result of an internal error. The compiler stops immediately after issuing this error message.

Recovery. Report the error to the GCSC.

```
LOWER DATA SEGMENT OVERFLOW
```

Cause. Run-time objects allocated in the lower half of the user data segment exceed 32,768 words.

Recovery. One or more of the following:

- Use the HIGHCOMMON compiler directive to move some or all the program's common blocks to the upper half of the user data segment.
- Use the LOWBUFFER or HIGHBUFFER directives to move some or all the runtime buffer pool to the upper half of the user data segment.
- Use the HIGHCONTROL directive to move the run-time control block to the upper half of the user data segment.
- Reduce the size of the run-time control block (see RUN-TIME CONTROL BLOCK OVERFLOW message).
- Use the LARGECOMMON compiler directive to move some or all the program's common blocks to the extended data segment.
- Use the LARGEDATA directive to move local arrays and RECORDs that are named in DATA or SAVE statements, to the extended data segment.

```
MIS-ALIGNMENT FORCED BY EQUIVALENCE ** name
```

Cause. An EQUIVALENCE specification attempts to force an entity to reside at two locations simultaneously.

Recovery. Correct the EQUIVALENCE specification.

```
MISPLACED STATEMENT CONTINUATION LINE
```

Cause. A FORTRAN statement continuation line was found where the initial line of a statement was expected. This could be at the beginning of a program unit or after a compiler directive line. The END statement cannot have continuation lines. Comment lines can be interspersed among the continuation lines of a statement, but compiler directives cannot.

Recovery. Correct the source program.

```
MISSING FORMAT ** label
```

Cause. An I/O statement refers to a FORMAT statement label but no FORMAT statement with that label exists in the program unit.

Recovery. Supply the missing FORMAT statement.

```
MISSING INDEX FOR ARRAY
```

Cause. An array element reference has too few subscripts.

Recovery. Correct the reference.

```
MISSING STATEMENT NUMBER ** label
```

Cause. Reference has been made to a statement label that does not appear in the program unit.

Recovery. Supply the missing statement label.

```
MORE THAN ONE STRUCTURE FOR THIS ITEM ** name
```

Cause. name has been specified as two different kinds of entities, such as a variable and a subroutine.

Recovery. Remove all but one specification.

```
MULTIPLE COMMON SPECIFICATION
```

Cause. An identifier appears in more than one COMMON statement.

Recovery. Remove it from all but one of the COMMON statements.

```
MUST BE AN INTRINSIC FUNCTION
```

Cause. An identifier in an INTRINSIC statement is not the name of an intrinsic function in HP FORTRAN.

Recovery. Correct the statement.

```
MUST BE TYPE CHARACTER TO HAVE SIZE
```

Cause. A noncharacter entity specifies a length.

Recovery. Remove the length specification or change the data type to CHARACTER.

```
MUST START WITH ALPHABETIC CHARACTER
```

Cause. A statement begins with a nonalphabetic character.

Recovery. Correct the statement.

```
NON EXISTENT I/O CONTROL ** keyword
```

Cause. The indicated control specifier in an I/O statement is not one of the keywords known to HP FORTRAN.

Recovery. Correct the statement.

```
NON NUMERIC STATEMENT LABEL
```

Cause. A nonnumeric character appears in the label field (columns 1 through 5) of the initial line of a statement.

Recovery. Remove or replace the character.

```
NOT A SIMPLE VARIABLE
```

Cause. A statement appears to be an assignment statement but the left side of the assignment is not a variable, an array element, a RECORD field, or a substring.

Recovery. Correct the statement.

```
NUMBER OF DIMENSIONS INCOMPATIBLE ** array-name
```

Cause. The number of dimensions in an array is specified in two different, but contradictory, declarations.

Recovery. Remove one declaration.

```
ONLY THE LAST DIMENSION MAY HAVE AN ASTERISK UPPER BOUND **
array-name
```

Cause. The array declarator has an asterisk for the upper bound of one of the array's dimensions other than the last (rightmost) dimension.

Recovery. Correct the array declarator.

```
PARAMETER MISMATCH
```

Cause. A reference to a generic intrinsic function has an argument that is not one of the data types allowed for that generic function.

Recovery. Pass the correct argument types to the intrinsic function.

```
PARAMETER MISMATCH ** number
```

Cause. One of the following:

- The argument in the indicated position is not the data type or kind of entity that is expected by the intrinsic function or statement function being referenced.
- The argument in the indicated position has backslashes (indicating pass by value) surrounding it, but the procedure being called expects this argument to be passed by reference.

Recovery. Make the arguments consistent.

```
PARAMETER TYPE MISMATCH
```

Cause. A name and its corresponding value in a PARAMETER statement have incompatible data types.

Recovery. Correct the statement.

```
PARAMETERS ARE INCOMPATIBLE
```

Cause. The number of actual arguments does not agree with the number of dummy arguments.

Recovery. Make the numbers of arguments consistent.

```
PARENTHESIS MISMATCH
```

Cause. The number of left parentheses in this statement is not equal to the number of right parentheses.

Recovery. Make the parentheses balance.

```
PRIMARY GLOBAL AREA OVERFLOW
```

Cause. The program has too many common blocks in the user data segment or refers to too many entities in such common blocks.

Recovery. Use the EXTENDCOMMON compiler directive to reduce the number of pointer words that must be allocated in the primary global data area (the first 256 words of the user data segment), or use the LARGECOMMON directive to move some or all the program's common blocks to the extended data segment, because they do not need any pointers in the primary global area.

```
PRIMARY LOCAL AREA OVERFLOW
```

Cause. This program unit references too many local variables and arrays and LARGECOMMON common blocks, so that the required pointers do not fit into the primary local data area on the run-time stack.

Recovery. Simplify the program unit.

```
PROCEDURE ALREADY DECLARED
```

Cause. This program unit has been declared previously.

Recovery. Remove one declaration.

```
PROGRAM UNIT TOO LARGE ** name
```

Cause. More than 32,767 words of object code (including executable instructions and read-only data such as constants and FORMAT statements) were generated for this program unit.

Recovery. Divide the program unit into smaller program units.

```
RECORD MAY ONLY BE EQUIVALENCED TO A RECORD
```

Cause. An attempt has been made to equivalence a RECORD, defined by RECORD declaration statements, to an entity that is not a RECORD.

Recovery. Correct the EQUIVALENCE statement.

```
RECORDS MAY ONLY HAVE ONE DIMENSION
```

Cause. An attempt has been made to declare a RECORD, or an array within a RECORD, with more than one dimension.

Recovery. Correct the statement.

```
RUN-TIME CONTROL BLOCK OVERFLOW
```

Cause. The run-time control block exceeds 32,767 words.

Recovery. Reduce one or more of the following values:

- The highest I/O unit number defined
- The total number of I/O units defined
- The OPEN, SYNCDEPTH, QDEPTH, or MAXREPLY parameters in the RECEIVE directive
- The STARTUP, ASSIGNS, or PARAM values in the SAVE directive

```
SOURCE DIRECTIVES NESTED TOO DEEPLY
```

Cause. A source input file being read by a SOURCE compiler directive contains another SOURCE directive, and the total nesting depth exceeds six levels. The compiler stops immediately after issuing this error message.

Recovery. Reorganize the source files.

```
SOURCE LIST OVERFLOW
```

Cause. A SOURCE compiler directive specifies too many section names. The compiler stops immediately after issuing this error message.

Recovery. Notify the GCSC.

```
STATEMENT FUNCTION NAME NOT UNIQUE
```

Cause. A statement function has the same name as another statement function or a variable. Note that any statement of the form

```
name (anything) = anything
```

is taken as a statement function definition if the name has not been declared as an array.

Recovery. Rename the statement function, or correct the statement if the name is a misspelled array name.

```
STATEMENT NOT ALLOWED IN BLOCK DATA
```

Cause. A statement other than a declarative statement or a DATA statement appears in a block data program unit. Note that RECORDs cannot be declared in block data program units because they cannot be initialized with DATA statements.

Recovery. Delete the invalid statement.

```
STATEMENT NOT ALLOWED IN LOGICAL IF
```

Cause. The dependent statement of a logical IF statement is a DO statement, another logical IF statement, or an END statement, none of which is allowed.

Recovery. Correct the statement.

```
STATEMENT NOT ALLOWED IN MAIN PROGRAM
```

Cause. A main program contains an ENTRY statement, which is not allowed.

Recovery. Correct the source program.

```
STATEMENT OUT OF ORDER
```

Cause. The FORTRAN rules for statement ordering within a program unit have been violated. For example, a declarative statement or a statement function definition follows an executable statement. Note that any statement of the form

```
name (anything) = anything
```

is taken as a statement function definition if the name has not been declared as an array.

Recovery. Correct the statement sequence, or correct the statement if the name is a misspelled array name.

```
STATEMENT TOO LONG
```

Cause. This statement contains more than 1320 characters. This can happen although you have not exceeded the limit of 19 continuation lines, if source lines are longer than 72 characters and you do not select the ANSI option.

Recovery. Correct the source.

```
STMT FN ARG MUST BE SIMPLE VARIABLE
```

Cause. A dummy argument of a statement function is not a simple variable.

Recovery. Change the argument to a simple variable.

```
STRING MAY NOT BE EMPTY
```

Cause. A zero-length character string was found.

Recovery. Correct the string.

```
STRING OVERFLOW
```

Cause. The title string in a PAGE directive contains more than 128 characters.

Recovery. Shorten the title.

```
SUBSCRIPT MUST BE FORMAL OR COMMON ** array-name
```

Cause. A dummy argument array in a subprogram is declared with an upper or lower bound that is a variable but is neither another dummy argument nor an item in a common block.

Recovery. Correct the source.

```
SUBSCRIPT MUST BE INTEGER VALUE
```

Cause. The upper or lower bound in an array declarator involves a constant or variable of a non-integer data type or an integer constant having a value outside the range -2,147,483,648 through 2,147,483,647. Or, an array element has a subscript expression of a non-integer data type.

Recovery. Correct the declarator or subscript expression.

```
SUBSCRIPT MUST BE SIMPLE VARIABLE
```

Cause. A dummy argument array is declared with an upper or lower bound that is not a dummy argument variable, a variable in a common block, a constant, or an expression involving only these.

Recovery. Correct the dimension bound expression.

```
SUBSTRING MUST BE A VARIABLE
```

Cause. The program attempts to use a substring of the value returned by a function. FORTRAN does not allow substrings of function references.

Recovery. Correct the statement.

```
SUBSTRING MUST BE TYPE CHARACTER
```

Cause. The program uses a substring of a variable or array element of a data type other than character.

Recovery. Correct the statement.

```
SUBSTRING TOO LARGE FOR ITEM
```

Cause. A substring has been specified with a length greater than that of the variable or array element.

Recovery. Correct the statement.

```
SYMBOL TABLE LOOKUP ERROR
```

Cause. Compiler error.

Recovery. Notify the GCSC.

```
SYNTAX ERROR
```

Cause. A source statement is syntactically incorrect.

Recovery. Correct the statement.

```
THERE CAN BE ONLY ONE RECORD PER COMMON BLOCK ** name
```

Cause. An attempt has been made to allocate more than one RECORD in the indicated common block.

Recovery. Correct the source.

```
THIS PROGRAM IS USED INCONSISTENTLY ** name
```

Cause. A function subprogram is used as a subroutine, or a subroutine is used as a function.

Recovery. Make subprogram definition and use consistent.

```
TOKEN OVERFLOW INTO UNSCANNED TEXT
```

Cause. The compiler’s scan buffer is full. A possible cause for this message is a DATA statement that contains too many entries.

Recovery. Notify the GCSC.

```
TOO MANY COMMON BLOCKS IN THIS PROGRAM UNIT
```

Cause. This program unit declares more than 64 different common blocks.

Recovery. Eliminate declarations of unused common blocks; combine common blocks that cannot be eliminated.

```
TOO MANY CONCATENATED OPERANDS
```

Cause. A character expression exceeds the HP FORTRAN limit of 64 concatenated operands.

Recovery. Concatenate in two steps, or simplify the expression.

```
TOO MANY CONTINUATION LINES
```

Cause. This statement has more than 19 continuation lines.

Recovery. Correct the statement.

```
TOO MANY DATA CONSTANTS
```

Cause. A constant list in a DATA statement has more items than the corresponding name list.

Recovery. Correct the statement.

```
TOO MANY ERRORS
```

Cause. The total number of error messages issued exceeds the limit specified in the ERRORS compiler directive, or 100 errors if an ERRORS directive is not specified. The compiler stops all work after issuing this message.

Recovery. Either increase the limit in the ERRORS compiler directive, or correct the errors already diagnosed.

```
TOO MANY I/O UNIT NUMBERS
```

Cause. The total number of different I/O unit numbers specified in the compilation exceeds the limit of 128, including units 4, 5, and 6 which are always defined.

Recovery. Reduce the number of I/O unit numbers that appear in UNIT compiler directives and as constant unit numbers in I/O statements throughout the source program.

```
TOO MANY PARAMETERS
```

Cause. A subprogram has more than 29 arguments.

Recovery. Reduce the number of arguments (by placing them in common, for example).

```
TWO COMMON AREAS EQUIVALENCED ** name
```

Cause. An EQUIVALENCE statement attempts to place the indicated entity into two different common blocks.

Recovery. Correct the EQUIVALENCE specification.

```
TWO DO RANGES OVERLAP
```

Cause. DO loop bodies partially overlap (neither is completely nested within the other).

Recovery. Correct the source.

```
TWO MAIN PROGRAM UNITS
```

Cause. The source program contains two main program units.

Recovery. Remove one main program unit.

```
TYPE COMPLEX VALUES CAN ONLY BE COMPARED FOR (IN)EQUALITY
```

Cause. An expression uses one of the comparison operators .LT., .LE., .GE., or .GT. to compare two values, one or both of which is of the complex data type. Only .EQ. and .NE. can be used for comparing such values.

Recovery. Correct the source program.

```
TYPE MISMATCH IN DATA STATEMENT ** name
```

Cause. The data type of a variable item in a DATA statement is incompatible with that of its corresponding constant item.

Recovery. Correct the statement.

```
UNABLE TO ALLOCATE EXTENDED SEGMENT FOR SYMBOL TABLE
```

Cause. FORTRAN was unable to compile your program because it could not obtain an extended memory segment for its symbol table. This usually means that your current default disk volume is nearly full. The compiler stops immediately after issuing this error message.

Recovery. Make more space available on the default disk volume, or use the PARAM SWAPVOL to make the compiler use a different disk volume that has more available space.

```
UNDELIMITED STRING
```

Cause. A character string has no terminating apostrophe.

Recovery. Correct the string.

```
UNRESOLVED DO OR BLOCK IF
```

Cause. No terminal statement was found for a DO loop body or a block IF statement sequence.

Recovery. Correct the source.

```
UPPER BOUND LESS THAN LOWER BOUND
```

Cause. The upper bound of an array is less than its lower bound.

Recovery. Correct the array declaration.

```
UPPER DATA SEGMENT OVERFLOW
```

Cause. Run-time objects allocated to the upper half of the user data segment have a total size exceeding 32,768 words of memory space.

Recovery. One or more of the following:

- Modify the HIGHCOMMON compiler directive to assign fewer common blocks to the upper half of the user data segment.
- Modify the LOWBUFFER or HIGHBUFFER directives to move some or all the run-time buffer pool to the lower half of the user data segment.
- Omit the HIGHCONTROL directive, to move the run-time control block to the lower half of the user data segment.
- Reduce the size of the run-time control block (see RUN-TIME CONTROL BLOCK OVERFLOW message).
- Use the LARGECOMMON compiler directive to move some or all the program's common blocks to the extended data segment.

```
VALUE OUT OF RANGE
```

Cause. One of the following:

- A compiler directive contains a constant whose value is outside the allowed range.
- An I/O unit number is a constant whose value is outside the range 1 through 999.
- A substring bound is a constant whose value is outside the range 1 through 255.

Recovery. Correct the source.

```
VARIABLE HAS NO VALUE STORED INTO IT ** name
```

Cause. The indicated variable is used in an executable statement, but is not defined in the same program unit.

Recovery. Correct the source.

```
WRONG I/O CONTROL FOR THIS STATEMENT
```

Cause. A control specifier is not defined for the I/O statement in which it appears (for example, FMT in an OPEN statement).

Recovery. Correct the statement.

Warning Messages

```
BACKSLASHES SUPERFLUOUS WITH GUARDIAN DIRECTIVE
```

Cause. You used backslashes around a pass-by-value argument in a call to a procedure that was declared in a GUARDIAN directive.

Recovery. Remove the backslashes.

```
BYTE ADDRESS CONVERTED TO WORD ADDRESS
```

Cause. You specified a pass-by-reference argument of type CHARACTER in a call to a procedure that was declared in a GUARDIAN directive, but the procedure expects an argument of a different type. The compiler converts the byte address of the CHARACTER argument into the word address required by the procedure. If the CHARACTER argument begins on a word boundary, the program passes the correct address; otherwise, the program passes the address of the character that precedes the first character in the CHARACTER argument.

Recovery. Replace the CHARACTER argument with a type INTEGER argument that contains the same value. You could do this by declaring the CHARACTER and INTEGER items, along with an appropriate EQUIVALENCE statement, within a RECORD declaration.

```
COMMON BLOCK ALREADY ALLOCATED DIFFERENTLY ** block-name
```

Cause. You named a common block on either a HIGHCOMMON or a LARGECOMMON directive, but FORTRAN has established a different allocation for that common block when it was declared in a previous program unit in the same compilation.

Recovery. Either remove the indicated common block name from the directive, or move the directive nearer to the beginning of the source input file.

```
CPU TNS DIRECTIVE WILL NOT BE SUPPORTED IN THE D00 RELEASE
```

Cause. The CPU TNS compiler directive, which causes the compiler to create an object program that can then be executed on a NonStop 1+ system, will no longer be supported in the D00 release.

Recovery. Change the program so that it can be executed on a NonStop system.

```
DEFINE NAME NOT PERMITTED IN DIRECTIVE ** directive
```

Cause. The indicated compiler directive specifies a file system DEFINE name in a context where only a Guardian file name is permitted. The compiler ignores the directive.

Recovery. Replace the DEFINE name with a file name.

```
DUPLICATE SECTION NAME ** name
```

Cause. Either of the following:

- The same section name appears twice in a single SOURCE directive. The section is included only once.
- Two SECTION directives with the same name appear in a single source file. The compiler ignores the second section.

Recovery. Correct the source.

```
EARLIER DIRECTIVE OVERRIDES ** name
```

Cause. A compiler directive tries to re-specify something that has already been specified by another directive that was processed earlier. The compiler ignores the current directive.

Recovery. Remove the ignored directive.

```
ERRORFILE DIRECTIVE IGNORED ** reason
```

Cause. The ERRORFILE compiler directive specifies a file name that could not be used for this purpose, for the reason indicated. The possible reasons include: source input file not EDIT format, missing file name, invalid file name, non-existent device name, not a disk file name, existing file is not entry-sequenced, existing file's filecode is not 106, and unable to purge existing file. The compiler proceeds as if the ERRORFILE directive were not present.

Recovery. Depending on the reason stated in the message, either delete the ERRORFILE compiler directive, or change the directive to specify a different file name, or purge the existing file named in the directive.

```
IDENTIFIER EXCEEDS 31 CHARACTERS
```

Cause. Characters beyond the 31st in a symbolic name are ignored.

Recovery. Change the name. This error is likely to cause additional errors if this name's first 31 characters are the same as another symbolic name's first 31 characters.

```
ILLEGAL OPTION SYNTAX
```

Cause. A compiler directive contains a syntax error. The directive is ignored.

Recovery. Correct the directive.

```
INCONSISTENT COMMON ALLOCATION ** block-name
```

Cause. You named the same common block on both a HIGHCOMMON and a LARGECOMMON directive. FORTRAN allocates storage for the common block based on the first directive that named that block.

Recovery. Remove the common block name from one of the two directives.

```
INITIAL STACK MARKER AT OR NEAR OVERFLOW
```

Cause. The object program has so much data allocated in the lower half of the user data segment that there is little space remaining for the run-time stack. The object program might run correctly, or it might get a “stack overflow” trap.

Recovery. Make more room in the lower half of the user data segment by using the same methods suggested for the LOWER DATA SEGMENT OVERFLOW error message.

```
INVALID FILE NAME
```

Cause. The file name in a SEARCH compiler directive has an incorrect form. The directive is ignored.

Recovery. Correct the directive.

```
INVALID OCTAL DIGIT
```

Cause. An octal constant contains an 8 or 9.

Recovery. Correct the constant.

```
MAIN PROCEDURE IS MISSING
```

Cause. The compilation does not include a main program unit. The object file is therefore not executable.

Recovery. Correct the source program or combine this object file with a separately compiled object file that does include a main program unit.

```
NONSTOP DIRECTIVE IGNORED UNLESS ENV=COMMON IS SPECIFIED
```

Cause. The NONSTOP directive is only valid if ENV COMMON has been specified. Programs that specify ENV OLD are always compiled for NonStop execution if any program unit contains a CHECKPOINT or START BACKUP statement.

Recovery. Add the ENV COMMON directive or remove the NONSTOP directive, as appropriate.

```
PARAMETER MISMATCH ** number
```

Cause. A reference to a procedure subprogram has an argument of a data type or entity kind that is incompatible with the corresponding argument in the procedure's declaration or in a previous reference to it.

Recovery. Make the arguments consistent.

```
PARAMETER MISMATCH ** name
```

Cause. A previous reference to this procedure subprogram had an argument of a data type or a kind of entity that is incompatible with the indicated dummy argument.

Recovery. Make the arguments consistent.

```
PREVIOUS ERROR/WARNING MESSAGES NOT INCLUDED IN ERRORFILE
```

Cause. An ERRORFILE compiler directive appeared after one or more error or warning messages had been issued. This message, which will be the first message in the error file, indicates that earlier messages exist in the main listing output file but not in the error file.

Recovery. Examine the earlier messages in the main listing output file, and either eliminate the causes of those messages, or move the ERRORFILE compiler directive nearer the beginning of the source input file.

```
PROCEDURE IS NOT A GUARDIAN ROUTINE ** proc-name
```

Cause. You used a GUARDIAN directive that named a procedure which is not a Guardian procedure, a Saved Message Utility procedure, or a FORTRAN utility routine. FORTRAN generates the normal FORTRAN calling sequence for calls to the named procedure.

Recovery. Correct the procedure name or remove it from the GUARDIAN directive.

```
PROCEDURE NOT FOUND ** proc-name
```

Cause. A CONSULT compiler directive specifies a procedure name that is not found in the designated object file. The compiler skips the missing procedure and carries on with the remainder of the directive.

Recovery. Remove the indicated procedure name from the CONSULT directive, or change the file name to that of a file that contains the specified procedure.

```
PROCEDURE INCOMPATIBLE WITH RUN-TIME ENVIRONMENT ** proc-name
```

Cause. A CONSULT directive references an object file which was compiled with a different setting for the ENV directive or for the NONSTOP directive.

Recovery. Recompile the source of the target of the CONSULT directive with the settings used in the current compilation or change the directives being used to match those of the previous compilation.

```
RECORD ALLOCATION POSSIBLY INCOMPATIBLE
```

Cause. Allocation of subrecords with non-character data changed from previous releases.

Recovery. Compare the storage map for the record with that produced by the previous version of FORTRAN used. If there are differences, add filler declarations to make the new allocation conform to the old one. This warning is issued only if the value of PARAM FORTRAN^RECORD^WARNING is nonzero.

```
SECTION NOT FOUND ** section-name
```

Cause. A section name mentioned in a SOURCE directive was not found in the source file.

Recovery. Check the source; check the spelling of section name.

```
SOURCE LINE TRUNCATED
```

Cause. A source line exceeded 132 characters in length. Characters beyond the 132nd are ignored.

Recovery. Correct the source.

```
SPECIFIED NUMBER OF DATA PAGES MAY BE INSUFFICIENT
```

Cause. The argument of a DATAPAGES directive is less than the compiler's estimate of the number of data pages required.

Recovery. Try a larger number.

```
START BACKUP STATEMENT ENCOUNTERED WITHOUT NONSTOP SPECIFIED
```

Cause. A START BACKUP statement appears in a compilation in which ENV COMMON is specified but NONSTOP is not specified. A program that specifies ENV COMMON must also specify NONSTOP in order to be able to run as a NonStop process.

Recovery. Add the NONSTOP directive to the beginning of the compilation.

```
UNDEFINED OPTION
```

Cause. An unrecognizable compiler directive was encountered. The directive is ignored.

Recovery. Correct the source.

```
VALUE OUT OF RANGE
```

Cause. A value in a compiler directive is too great or too small. The directive is ignored.

Recovery. Correct the source.

G

Run-Time Diagnostic Messages

FORTRAN displays run-time diagnostic messages for the following types of errors:

- I/O errors
- Intrinsic function errors
- Common run-time environment messages

Topics covered in this section include:

The first part of this appendix describes the features of FORTRAN statements and utility routines that enable FORTRAN programs to handle run-time errors.

The second part of this appendix describes the format and content of the diagnostic messages that the FORTRAN run-time environment writes to your log file. It describes the format of messages for programs compiled with ENV OLD in effect as well as for programs compiled with ENV COMMON in effect.

I/O Errors

FORTRAN statements and utility routines that call Guardian system procedures might report run-time errors.

Most of the statements during which an error can occur include the IOSTAT = *ios* and ERR = *label* parameters that enable your program to detect the error and continue running.

ios is an integer variable or integer array element in which FORTRAN returns an error number if an error occurs while the statement or routine is executing. If an error does not occur, the statement or routine sets *ios* to zero. For more information about the error numbers returned in *ios*, see the Error Numbers on page 6-5.

label is the label of an executable statement in the current program unit to which FORTRAN transfers control if an error occurs while the statement or routine is executing.

If you specify *ios* but not *label*, your program continues executing with the FORTRAN statement that follows the statement during which the error occurred. You

can determine if an error occurred and, if so, its error number, by analyzing the value in *ios*.

If you specify *label* but not *ios*, your program continues executing with the statement labeled *label*. Although your program can detect that an error occurred—by the fact that it is now executing at a specified label—you cannot determine the specific error that occurred.

If you specify both *label* and *ios*, FORTRAN transfers control to *label* and you can determine the error that occurred by the error number in *ios*.

If you do not specify *label* or *ios*, FORTRAN terminates your program and writes an error message to the log file.

FORTRAN utility routines include a positional parameter in which the routine stores an error number. If an error occurs and you do not specify the error parameter, FORTRAN terminates your program and writes an error message to the log file.

START BACKUP and CHECKPOINT Errors

Both the START BACKUP and CHECKPOINT statements include a BACKUPSTATUS= specifier, as well as an ERR= specifier. These specifiers serve the same purpose for START BACKUP and CHECKPOINT statements as IOSTAT and ERR do for I/O statements.

If ENV OLD is in effect and an error occurs when your program executes a START BACKUP or CHECKPOINT statement that does not include a BACKUPSTATUS= specifier, the FORTRAN run-time library writes a message to your terminal. Your program continues running either with the executable statement designated in the ERR= specifier, or, if you do not specify ERR=, with the executable statement following the START BACKUP or CHECKPOINT statement.

If ENV COMMON is in effect and an error occurs when your program executes a START BACKUP or CHECKPOINT statement that does not include a BACKUPSTATUS= specifier, the FORTRAN run-time library writes a message to the standard log file. If the START BACKUP or CHECKPOINT statement includes an ERR= specifier, your program continues running at the label specified in the ERR= specifier. Otherwise, the run-time library terminates your program.

Intrinsic Errors

The following intrinsic functions can cause run-time errors:

- ACOS, DACOS, ASIN, and DASIN: argument value is less than -1.0 or greater than 1.0.
- ALOG, DLOG, ALOG10, DLOG10: argument less than or equal to zero. \
- ATAN2 and DATAN2: both arguments are zero.
- CLOG: argument value is CMPLX (0, 0).
- SQRT and DSQRT: argument value is less than zero.
- DSQRT when called by CSQRT: if the argument to CSQRT is z = CMPLX (x, y), the error can occur when x > CABS (z).

Error Messages

If an error occurs while your program is running, the FORTRAN run-time library writes a diagnostic message to the log file. The format of the diagnostic message depends on the nature of the error, and whether or not you specify ENV OLD or ENV COMMON when you compile your program.

Diagnostic Messages With ENV OLD

If you compile your program with ENV OLD, the FORTRAN run-time library writes messages in one of three different formats, depending on the statement or function that causes the error. The three formats correspond to whether the error occurs while executing:

- A READ or WRITE statement
- An I/O statement other than a READ or WRITE statement
- An intrinsic function

READ and WRITE Message Format

The formatter writes an error message to the log file in the following format if either the formatter or the Guardian file system detects an error while executing a READ or WRITE statement:

```
FORMATTER ERROR @ p-addr, space. segment
UNIT uuu, ERROR errno
```

p-addr

is the program location, in octal, where the error was detected, as shown by the instruction address in the P-register.

space

is the two-letter code space identifier as follows:

```
UC User code                SC System code
UL User library             SL System library
```

segment

is the segment number, in octal, within *space*.

uuu

is the number of the unit specified in the READ or WRITE statement. Unit number 000 indicates an error on $RECEIVE; an all blank unit number refers to an internal file.

errno

is a 5-digit number that identifies the I/O error.

Errors 00001 through 00255 are file-system error numbers. For more information about file system error numbers, see the *Guardian Procedure Errors and Messages Manual* . You can get a brief description of file system errors by entering the following TACL command:

```
ERROR error-number
```

Errors 00256 through 00274 are unique to the formatter. See Formatter Run-Time Messages on page G-8.

System Error Message Format

The FORTRAN run-time-library writes an error message to the log file in the following format if either the run-time library or the Guardian file system detects an error while executing an I/O statement, other than a READ or WRITE statement, or START BACKUP or CHECKPOINT statement:

```
operation ERROR @ space. seg p-addr IN process-id
UNIT uuu, ERROR errno
```

operation

is one of:

BACKSPACE	ENDFILE	OPEN	REWIND
CHECKPOINT	FORTRANSPOOLSTART	POSITION	STARTBACKUP
CLOSE	INQUIRE		

space

is the two-letter code space identification as follows:

UC User code	SC System code
UL User library	SL System library

seg

is the segment number, in octal, within *space*.

p-addr

is the program location, in octal, where the error was detected, as shown by the instruction address in the P-register.

process-id

is the identification of the process in which the error occurred. It has the form

[\ *node* .] [*process-name*] (*cpu* , *pin*)

\ *node*

is the node name, if the process is running on the same node as the home terminal.

process-name

is the process name, if the process has a name.

cpu

is the processor number in which the process runs.

pin

is the process id number within the process's processor.

uuu

is the number of the unit specified statement. Unit number is 000 for a START BACKUP or CHECKPOINT statement, and for an INQUIRE statement if a unit number is not specified.

errno

is a 5-digit number that identifies the error condition.

Intrinsic Error Message Format

The FORTRAN run-time-library unconditionally terminates your program and writes an error message to the log file in the following format if an intrinsic function detects an error.

Run-time intrinsic function diagnostic messages have the general form:

```
FORTRAN LIBRARY CALL ERROR: name
```

name

is the name of the intrinsic function that detected the error.

Diagnostic Messages With ENV COMMON

If you compile your program with ENV COMMON, the FORTRAN run-time library writes all diagnostic messages to the standard log file in the same format, regardless of the cause of the error.

Message Format

This subsection describes the format of the messages that FORTRAN writes to the log file. If you compile your program with ENV COMMON, the run-time library writes all diagnostic messages in the same format, regardless of the type of error that occurred.

```
process_name - *** Run-time Error nnn ***
process_name - message [ ( additional_information ) ]
[ process_name - optional_text ]
process_name - From: top_of_stack
process_name - : : : :
process_name - bottom_of_stack
```

process_name

identifies the process in which the error occurred.

nnn

is the number of the diagnostic message.

message

is the text associated with message number *nnn*.

additional_information

if present, gives more detail about message. For example, if an error occurs while accessing a file, *additional_information* might be the file-system error number.

optional_text

if present, provides additional information about the error. For example, it might show the FORTRAN unit number. *optional_text* helps you identify the cause of the error.

top_of_stack

shows the name of the procedure that invoked the run-time library routine in which the error was detected, the offset within the procedure, and the number of the code segment in which the procedure's code is located.

bottom_of_stack

shows the name of the first procedure—the main procedure—of the process in which the error occurred, the offset within the procedure, and the number of the code segment in which the procedure's code is located. The stack trace includes all procedures between *top_of_stack* and *bottom_of_stack*.

The following examples show messages that FORTRAN might write to the standard log file:

- If a program passes a negative value to a square root function, FORTRAN writes a message such as the following to the standard log file:

```
\NODE.$Z012:3 - *** Run-time Error 049 ***
\NODE.$Z012:3 - Square root domain fault
\NODE.$Z012:3 - From: DRAWIT + %513, UC.00
\NODE.$Z012:3 - CIRCLE + %21, UC.00
\NODE.$Z012:3 - MYPROG + %7, UC.00
```

- If a program tries to open standard input but the file does not exist, FORTRAN writes a message such as the following to the standard log file:

```
\NODE.$Z012:3 - *** Run-time Error 059 ***
\NODE.$Z012:3 - Standard input file error (11)
\NODE.$Z012:3 - From: READREC +%54, UC.00
\NODE.$Z012:3 NEXTREC + %15, UC.00
\NODE.$Z012:3 - COMPUTE + %214, UC.00
\NODE.$Z012:3 - MYPROG + %7, UC.00
```

  Note that the error message includes the number of the file-system error number (11).

- If your FORTRAN program cannot create a new file, FORTRAN writes a message such as the following to the standard log file. Note that the message includes an informational line that shows the unit associated with the file and the external Guardian file name:

```
\NODE.$TEST:8403781 - *** Run-time Error 082 ***
\NODE.$TEST:8403781 - GUARDIAN I/O error 14
\NODE.$TEST:8403781 - Unit 008 = $FOO.A.B
\NODE.$TEST:8403781 - From SUB2 + %111, UC.00
\NODE.$TEST:8403781 - SUB1 + %2, UC.00
\NODE.$TEST:8403781 - MAIN^ + %7, UC.00
```

Formatter Run-Time Messages

If an error occurs during the execution of a READ or a WRITE statement, the formatter writes a message to the log file. Although the message numbers are the same, whether your program specifies ENV OLD or ENV COMMON, the message text depends on which ENV parameter you specify.

Formatter Messages with ENV COMMON

FORTRAN writes the following messages to the standard log file if you specify ENV COMMON when you compile your program. (The next subsection shows the error text that the formatter writes if your program specifies ENV OLD.)

255

```
OLD routine not executable in COMMON environment
```

Cause. A program called a C-series routine that was compiled without an ENV directive or a D-series or later library routine that was either compiled without an ENV directive or with the ENV OLD compiler directive, such as FORTRSNSPOOLSTART or FORTRANCOMPLETION, while running with the CRE.

Effect. The run-time library terminates the program.

Recovery. Do one of the following:

- Compile your program with an ENV OLD compiler directive
- Remove the specification to an ENV directive
- Change the call to a routine appropriate for the COMMON environment.

256

```
Unknown unit value nnn
```

Cause. A routine referenced a unit that was not specified in a UNIT compiler directive or as a constant expression in the unit specifier of an I/O statement.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates your program.

Recovery. Specify the unit number in a UNIT compiler directive or use a constant expression in the unit specifier of the I/O statement, and recompile your program.

257

```
Invalid parameter value
```

Cause. An I/O statement contains an illegal parameter or parameter combination. For example, in a WRITE statement, UNLOCK=.TRUE. is invalid unless UPDATE is also present with a true value.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates your program.

Recovery. Correct the parameter combination.

258

```
OPEN specifies SCRATCH status and file name
```

Cause. An OPEN statement specifies STATUS='SCRATCH' as well as a subvolume name, a file name, or both.

Effect. File cannot be opened.

Recovery. A file opened with STATUS='SCRATCH' can either not specify a location for the file, or can specify a volume name but neither a subvolume name nor a file name.

Change the OPEN statement to either not specify a file name or to specify only a volume name:

```
OPEN (60, STATUS='SCRATCH') <-- OK
OPEN (60, STATUS='SCRATCH', FILE= '$U1') <-- OK
OPEN (60, STATUS='SCRATCH', FILE= '$U1.VOL') <-- ERROR
OPEN (60, STATUS='SCRATCH', FILE= '$U1.VOL.SCR1') <-- ERROR
```

259

```
OPEN for unit open with inconsistent attribute values
```

Cause. The attributes specified in an OPEN statement that referenced an already-open file were incompatible with the attributes in effect from the file's original open. If a routine opens a file connection for an already-open file, the attributes of the second (and subsequent) opens must be the same as those of the original open except for the BLANK parameter, which can be different.

Effect. The file remains open. The value of BLANK for the already-open file is not changed. If the OPEN statement specified an ERR or an IOSTAT clause, the program retrains control. Otherwise, the run-time library terminates the program.

Recovery. Modify the routine to ensure that multiple connections to the same file open specify the same attributes—or do not specify attributes—except, possibly, the BLANK parameter.

260

```
START BACKUP failed with status code nnn
```

Cause. A routine invoked START BACKUP but the backup could not be started. *nnn* specifies the reason the backup was not able to start. For more details, see Table 7-10 on page 7-102.

Effect. *nnn* gives the reason that the backup could not be created. *nnn* is a NonStop process status code. See Table 7-10 on page 7-102.

Recovery. Control returns to the routine, which might either retry the START BACKUP statement or run without a backup process.

261

```
CHECKPOINT failed with status code nnn
```

Cause. A routine called CHECKPOINT but the checkpoint request could not be completed. *nnn* specifies the reason the checkpoint was unsuccessful. For more details, see Table 7-10 on page 7-102.

Effect. *nnn* gives the reason the checkpoint was unsuccessful. nnn is a NonStop process status code. See Table 7-10 on page 7-102.

Recovery. Control returns to the routine, which might retry the CHECKPOINT statement, run without checkpointing, or terminate.

267

```
Buffer overflow for unit nnn
```

Cause. A routine tried to transfer a record that is larger than the maximum record length specified for the unit. The maximum record length is established at the time a unit is connected to a file. For more details, see the OPEN Statement on page 7-70.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates your program.

Recovery. Specify a larger buffer size either in a UNIT directive or in a TACL ASSIGN command.

270

```
Format loopback for unit nnn
```

Cause. The formatter encountered a right parenthesis with more data remaining to be formatted. However, no data was written as a result of the formats within the current set of parentheses. In the following example, the WRITE statement never completes because there are two values to write, `I` and `J`. The value of `I` is output but the formatter would repeatedly scan the 2X specification and the value of `J` would never be written:

```
WRITE(6, 100) i, j
100 FORMAT(1X, I4, (2X))
```

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates the program.

Recovery. Modify your FORMAT statement to account for all data specified in the WRITE statement.

271

```
Edit item mismatch for unit nnn
```

Cause. An item in the data list of an I/O statement is incompatible with its corresponding actual value or format edit descriptor. For example, a data item is a REAL value but the current item descriptor expects an integer:

```
WRITE(6, 100) r
100 FORMAT(1X, I4)
```

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates the program.

Recovery. Modify your FORMAT statement to specify the correct type for each data item specified in the WRITE statement.

272

```
Illegal input character for unit nnn
```

Cause. A numeric field contains a character other than 0 through 9, a decimal point, a comma, or a blank.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates the program.

Recovery. Correct the input data or change the program to describe correctly the data the program is reading.

273

```
Illegal format for unit nnn
```

Cause. A format specification is syntactically correct but is used in a context in which it is not valid. For example, the specification F5.5 is valid in a format for input but not for output.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates the program.

Recovery. Change the format specification so that each element of the format is correct for the context in which it is used.

274

```
Numeric overflow for unit nnn
```

Cause. An input value is too large for the variable in which it was to be stored.

Effect. If the program executed an I/O statement that includes an ERR clause or an IOSTAT clause, the program retains control. Otherwise, the run-time library terminates the program.

Recovery. Change the variable that receives the data to accept a larger value or change the data the program reads to a value that does not exceed the size of the variable into which it is being read.

Formatter Messages With ENV OLD

FORTRAN displays the following diagnostic messages if a run-time error occurs and you specify ENV OLD when you compile your program. See the preceding subsection for cause, effect, and recovery information.

256

```
BAD UNIT
```

257

```
BAD PARAMETER
```

258

```
not used with ENV OLD
```

259

```
not used with ENV OLD
```

260

```
not used with ENV OLD
```

261

```
not used with ENV OLD
```

267

```
BUFFER OVERFLOW
```

270

```
FORMAT LOOPBACK
```

271

```
EDIT ITEM MISMATCH
```

278

```
ILLEGAL INPUT CHARACTER
```

273

```
ILLEGAL FORMAT
```

274

```
NUMERIC OVERFLOW
```

System Messages

The FORTRAN run-time library displays the messages in this and the following subsections if an error occurs while executing a statement or intrinsic routine. The error might be detected by the operating system, by routines in the FORTRAN run-time environment, or by routines in common run-time environment.

System Messages With ENV OLD

If your program specifies ENV OLD and an error occurs while running your program, the FORTRAN run-time library writes a message to your log file according to the format described earlier in this section. However, the message specifies only the file system error number that occurred—there is no message text.

System errors can occur in programs that specify ENV OLD on the following statements and utility routines:

BACKSPACE	FORTRANSPOOLSTART	POSITION
CHECKPOINT	INQUIRE	REWIND
CLOSE	OPEN	STARTBACKUP
ENDFILE		

System Messages With ENV COMMON

If your program specifies ENV COMMON and an error occurs while running your program, the FORTRAN run-time library writes a message to the standard log file according to the format described earlier in this section. The following subsections show the text of the messages that FORTRAN writes.

System errors can occur in programs that specify ENV COMMON on the following statements and utility routines:

BACKSPACE	FORTRAN_SETMODE_	READ
CHECKPOINT	FORTRAN_SPOOL_OPEN_	REWIND
CLOSE	INQUIRE	START BACKUP
ENDFILE	OPEN	WRITE
FORTRAN_CONTROL_	POSITION	

Trap Messages

The run-time library reports the messages in this subsection if a trap occurs. The runtime library terminates your program.

The run-time treats trap 4, arithmetic fault, as a program logic error, not a trap.

1

```
Unknown trap
```

The run-time trap processing function was called with an unknown trap number.

2

```
Illegal address reference
```

An address was specified that was not within either the virtual code area or the virtual data area allocated to the process.

3

```
Instruction failure
```

An attempt was made to:

- Execute a code word that is not an instruction.
- Execute a privileged instruction by a nonprivileged process.
- Reference an illegal extended address.

4

```
Arithmetic fault
```

The overflow bit in the environment-register, ENV.<10>, was set to 1 for one of the following reasons:

- The result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type.
- An division operation was attempted with a zero divisor.

5

```
Stack overflow
```

A stack overflow fault occurs if:

- An attempt was made to execute a procedure or subprocedure whose local or sublocal data area extends into the upper 32K of the user data area.
- There was not enough remaining virtual data space for an operating system procedure to execute.

The amount of virtual data space available is G[0] through G[32767].

Operating system procedures require approximately 350 words of user-data stack space to execute.

6

```
Process loop-timer timeout
```

The new time limit specified in the latest call to SETLOOPTIMER has expired.

7

```
Memory manager read error
```

An unrecoverable read error occurred while the program was trying to bring in a page from virtual memory.

8

```
Not enough physical memory
```

This fault occurs for one of the following reasons:

- A page fault occurred, but there were no physical memory pages available for overlay.
- Disk space could not be allocated while the program is using extensible segments.

9

```
Uncorrectable memory error
```

An uncorrectable memory error occurred.

Run-Time Core Messages

The CRE writes the messages in this subsection if an error occurs during its own processing or if it receives a request from a run-time library to report a specific message.

11

```
Corrupted environment
```

Cause. Run-time data is invalid.

Effect. The run-time environment invokes PROCESS_STOP_, specifying the ABEND variant and the text “Corrupted environment.”

Recovery. The program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for run-time data. Check the program’s logic. Use Inspect to help isolate the problem or consult your system administrator.

12

```
Logic error
```

Cause. The run-time library detected a logic error within its own domain. For example, although each data item it is using is valid, the values of the data items are mutually inconsistent.

Effect. The run-time environment invokes PROCESS_STOP_, specifying the ABEND variant and the text “Logic error.”

Recovery. The program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for run-time library data. Check the program's logic. Use Inspect to help isolate the problem or consult your system administrator.

13

```
MCB pointer corrupt
```

Cause. The pointer at location G[0] of the program's user data segment to its primary data structure—the Master Control Block (MCB)—does not point to the MCB.

Effect. The run-time attempts to restore the pointer at G[0] and to write a message to the standard log file. However, because its environment might be corrupted, the runtime might not be able to log a message. In that case, it calls PROCESS_STOP_, specifying the ABEND variant, and the text "Corrupted Environment".

Recovery. Check the program's logic to see if it overwrote the MCB pointer at G[0]. Use Inspect to help isolate the problem. For details of how to determine where the program overwrites G[0], see Using Inspect on page 11-10.

14

```
Premature takeover
```

Cause. The backup process received a Guardian message that it had become the primary process but it had not yet received all of its initial checkpoint information from its predecessor primary process.

Effect. The run-time invokes PROCESS_STOP_, specifying the ABEND variant and the text "Premature takeover."

Recovery. If the takeover occurred because of faulty program logic, correct the program's logic. If the takeover occurred for other reasons, such as a hardware failure, you might want to rerun the program, provided that doing so will not duplicate operations already performed, such as updating a database a second time.

15

```
Checkpoint list inconsistent
```

Cause. A list of checkpoint item descriptors that the run-time maintains for NonStop processes was invalid.

Effect. The run-time terminates the program.

Recovery. The list of items to checkpoint is maintained in the program's address space. Check the program's logic. The program might have overwritten the checkpoint list. Use Inspect to help isolate the problem.

16

```
Checkpoint list exhausted
```

Cause. The run-time did not have enough room to store all the checkpoint information required by the program.

Effect. Program behavior is language and application dependent.

Recovery. Increase the checkpoint list object's size. For the routine that allocates your checkpoint list, see the language manual.

17

```
Cannot obtain control space
```

Cause. The run-time library could not obtain heap space for all of its data.

Effect. If the request came from the CRE, it terminates the program. Otherwise, program behavior is language and application dependent.

Recovery. You might be able to increase the amount of control space available to your program by reducing the number of files your program has open at the same time or by decreasing the size of buffers allocated to open files.

18

```
Extended Stack Overflow
```

Cause. A module could not obtain sufficient extended stack space for its local data.

Effect. Program behavior is language and application dependent.

Recovery. Increase the extended stack's size. For the routine that caused the extended stack overflow and for details on increasing the size of the extended stack, see the language manual.

20

```
Cannot utilize filename
```

Cause. A string, expected to be a valid file name, could not be manipulated as a Guardian external file name.

Effect. If the file name came from the CRE, the program is terminated. Otherwise, program behavior is language and application dependent.

Recovery. Check that the file names in the program are valid Guardian file names.

21

```
Cannot read initialization messages ( error )
```

Cause. During program initialization, the run-time could not read all the messages (start-up message, PARAM message, ASSIGN messages, and so forth) it expected from the file system. *error* is the file system error number the run-time received when it couldn't read an initialization message.

Effect. The run-time terminates the program.

Recovery. Consult your system administrator.

22

```
Cannot obtain program filename
```

Cause. The run-time could not obtain the name of the program file from the operating system.

Effect. The run-time terminates the program.

Recovery. Consult your system administrator.

23

```
Cannot determine filename ( error )
program_name.logical_name
```

Cause. The run-time could not determine the physical file name associated with *program_name.logical_name*.

Effect. The run-time terminates the program.

Recovery. Correct the *program_name.logical_name* and rerun your program. For general information on ASSIGN commands, see the *TACL Reference Manual*. For more information on using ASSIGNs, see the reference manual for your program's 'main' routine.

24

```
Conflict in application of ASSIGN
program_name.logical_name
```

Cause. ASSIGN values in your TACL environment conflict with each other. For example:

```
ASSIGN A, $B1.C.D
ASSIGN *.A, $B2.C.D
```

The first ASSIGN specifies that the logical name A can appear in no more than one program file. The second assign specifies that the name A can appear in an arbitrary number of program files. The run-time cannot determine whether to use the file C.D on volume $B1 or on volume $B2.

Effect. The run-time terminates the program.

Recovery. Correct the ASSIGNs in your TACL environment. For more information on using ASSIGNs, see the *TACL Reference Manual*.

25

```
Ambiguity in application of ASSIGN
logical_name
```

Cause. Your TACL environment specifies an ASSIGN such as:

```
ASSIGN A, $B1.C.D
```

but the program contains more than one logical file named A.

Effect. The run-time terminates the program.

Recovery. Correct the ASSIGNs in your TACL environment. For more information on using ASSIGNs, see the *TACL Reference Manual*.

26

```
Invalid PARAM value text ( error )
PARAM name ' value'
```

Cause. A PARAM specifies a value that is not defined by the run-time. For example, the value for a DEBUG PARAM must be either ON or OFF:

```
PARAM DEBUG [ ON  ]
            [ OFF ]
```

The run-time reports this error if a DEBUG PARAM has a value other than ON or OFF. *error*, if present, is a file-system error.

Effect. The run-time terminates the program.

Recovery. Modify the PARAM text and rerun the program. For more information on using PARAMs, see the *TACL Reference Manual*.

27

```
Ambiguity in application of PARAM
PARAM name ' value'
```

Cause. A PARAM specifies a value that is ambiguous in the current context. For example, the PARAM specification:

```
PARAM PRINTER-CONTROL A
```

is ambiguous if the program contains more than one logical file named A.

Effect. The run-time terminates the program.

Recovery. Correct the PARAM in your TACL environment. For more information on using PARAMs, see the *TACL Reference Manual*.

28

```
Missing language run-time library -- language
```

Cause. The run-time library for a module that is written in *language* is not available to the program.

Effect. The run-time terminates the program.

Recovery. Consult your system administrator.

29

```
Program incompatible with run-time library -- language
```

Cause. The language compiler used features that are not supported by the *language* run-time library that the program used.

Effect. The run-time terminates the program.

Recovery. Use a compiler and run-time library that are compatible. You might need to consult your system administrator.

Intrinsic Error Messages

Run-time libraries report the messages in this subsection if an error is detected in a math function.

40

```
Invalid function parameter
```

Cause. A function detected a problem with its parameters.

Effect. Program behavior is language and application dependent.

Recovery. Correct the parameter you are passing.

41

```
Range fault
```

Cause. An arithmetic overflow or underflow occurred while evaluating an arithmetic function.

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass values to the arithmetic functions that do not cause overflow.

42

```
Arccos domain fault
```

Cause. The parameter passed to the arccos function was not in the range:

$-1.0 \leq parameter \leq 1.0$

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass a valid value to the arccos function.

43

```
Arcsin domain fault
```

Cause. The parameter passed to the arcsin function was not in the range:

$-1.0 \leq parameter \leq 1.0$

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass a valid value to the arcsin function.

44

```
Arctan domain fault
```

Cause. Both of the parameters to an arctan2 function were zero. At least one of the parameters must be nonzero.

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass the correct value to the arctan2 function.

46

```
Logarithm function domain fault
```

Cause. The parameter passed to a logarithm function was less than or equal to zero. The parameter to a logarithm function must be greater than zero.

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass a valid value to the logarithm function.

47

```
Modulo function domain fault
```

Cause. The value of the second parameter to a modulo function was zero. The second parameter to a modulo function must be nonzero.

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass a nonzero value to the modulo function.

48

```
Exponentiation domain fault
```

Cause. Parameters to a Power function were not acceptable. Given the expression

x^y

the following parameter combinations produce this message:

$x = 0$ and $y \leq 0$

$x < 0$ and y is not an integral value

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass values that do not violate the above combinations.

49

```
Square root domain fault
```

Cause. The parameter to a square root function was a negative number. The parameter must be greater than or equal to zero.

Effect. Program behavior is language and application dependent.

Recovery. Modify the program to pass a nonnegative value to the square root function.

55

```
Missing or invalid parameter
```

Cause. A required parameter is missing or too many parameters were passed.

Effect. Program behavior depends on the function that was called and the language in which it is written.

Recovery. Correct the program to pass a valid parameter.

56

```
Invalid parameter value
```

Cause. The value passed as a procedure parameter was invalid.

Effect. Program behavior depends on the function that was called and the language in which it is written.

Recovery. Correct the program to pass a valid parameter value.

57

```
Parameter value not accepted
```

Cause. The value passed as a procedure parameter is not acceptable in the context in which it is passed. For example, the number of bytes in a write request is greater than the number of bytes per record in the file.

Effect. Program behavior depends on the function that was called and the language in which it is written.

Recovery. Correct the program to pass a valid parameter.

Input/Output Messages

The run-time reports the messages in this subsection if an error occurs when calling an I/O function.

59

```
Standard input file error ( error )
```

Cause. The file system reported an error when a routine tried to access the standard input file. *error* is a file-system error code.

Effect. The run-time can report this error when it closes your input file. All other instances are language and application dependent.

Recovery. If the error was caused by a read request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a read request from the run-time, consult your system administrator.

60

```
Standard output file error ( error )
```

Cause. The file system reported an error when the run-time called a system procedure to access standard output. *error* is the file-system error.

Effect. The run-time can report this error when it closes your output file. All other instances are language and application dependent.

Recovery. If the error was caused by a write request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a write request from the run-time, consult your system administrator.

61

```
Standard log file error ( error )
```

Cause. The file system reported an error when the run-time called a file system procedure to access the standard log file. *error* is the file-system error.

Effect. The run-time terminates your program.

Recovery. Consult your system administrator.

62

```
Invalid GUARDIAN file number
```

Cause. A value that is expected to be a Guardian file number is not the number of an open file.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

63

```
Undefined shared file
```

Cause. A parameter was not the number of a shared file where one was expected.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

64

```
File not open
```

Cause. A request to open a file failed because the file device is not supported.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

65

```
Invalid attribute value
```

Cause. A parameter to an open operation was not a meaningful value.

Effect. Program behavior is application dependent.

Recovery. Consult your system administrator.

66

```
Unsupported file device
```

Cause. The run-time received a request to access a device that it does not support.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

67

```
Access mode not accepted
```

Cause. The value of the *access* parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to open a spool file for input.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

68

```
Nowait value not accepted
```

Cause. The value of the *no_wait* parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to specify a nonzero value for *no_wait* for a device that does not support nowait operations.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

69

```
Syncdepth not accepted
```

Cause. The value of the *sync_receive_depth* parameter to an open operation was not valid in the context in which it was used. For example, it is not valid to specify a *sync_receive_depth* greater than one for a shared file.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

70

```
Options not accepted
```

Cause. The value of an open operation *options* parameter was not valid in the context in which it was used.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

71

```
Inconsistent attribute value
```

Cause. A routine requested a connection to a shared file that was already open, and the attributes of the new open request conflict with the attributes specified when the file was first opened.

Effect. Program behavior is language and application dependent.

Recovery. If your program supplied the attribute values, correct and rerun your program. Otherwise, consult your system administrator.

75

```
Cannot obtain buffer space
```

Cause. A routine was not able to obtain buffer space.

Effect. Program behavior is language and application dependent.

Recovery. None

76

```
Invalid external filename ( error )
```

Cause. A value that was expected to be a Guardian external file name is not in the correct format.

Effect. Program behavior is language and application dependent.

Recovery. If you supplied an invalid file name, correct the file name and rerun your program. Otherwise, consult your system administrator.

77

```
EDITREADINIT failed ( error )
```

Cause. A call to EDITREADINIT failed. *error*, if present, gives the reason for the failure. Possible values of *error* are:

Error Code	Error Name
-1	END-OF-FILE encountered
-2	I/O error
-3	Text file format error
-6	Invalid buffer address

Effect. Program behavior is language and application dependent. For more information, see the *Guardian Procedure Calls Reference Manual*.

Recovery. Recovery is language and application dependent.

78

```
EDITREAD failed ( error )
```

Cause. A call to EDITREAD failed. *error*, if present, gives the reason for the failure. Possible values of *error* are:

Error Code	Error Name
-1	END-OF-FILE encountered
-2	I/O error
-3	Text file format error
-4	Sequence number error
-5	Checksum error
-6	Invalid buffer address

Effect. Program behavior is language and application dependent.

Recovery. For more information, see the *Guardian Procedure Calls Reference Manual*.

79

```
OpenEdit failed ( error )
```

Cause. A call to OPENEDIT_ failed. *error*, if present, is the error returned by OPENEDIT_. A negative number is a format error. A positive number is a file-system error number.

Effect. Program behavior is language and application dependent.

Recovery. For more information, see the *Guardian Procedure Calls Reference Manual*.

80

```
Spooler initialization failed ( error )
```

Cause. An initialization operation to a spooler collector failed. *error*, if present, is the file system error code returned by the SPOOLSTART procedure.

Effect. Program behavior is language and application dependent.

Recovery. For more information, see the *Spooler Programmer's Guide*.

81

```
End of file
```

Cause. A routine detected an end-of-file condition.

Effect. Program behavior is language and application dependent.

Recovery. Correct your program to allow for an end-of-file condition or ensure that your program can determine when all the data has been read.

82

```
Guardian I/O error nnn
```

Cause. An operating system routine returned error *nnn*. This error is usually reported as a result of an event that is beyond control of your program such as a path or system is not available.

Effect. Program behavior is language and application dependent.

Recovery. Consult your system administrator.

90

```
Open conflicts with open by other language
```

Cause. Routines written in two different languages—for example, COBOL85 and FORTRAN—attempted to open a connection to a file using the same logical name. This error is reported only for nonshared files.

Effect. Program behavior is language and application dependent.

Recovery. Modify your program to use different logical names or coordinate logical names between the two routines so that they do not open the same logical file at the same time.

91

```
```

Spooler job already started

Cause. A FORTRAN routine attempted to open a spooler but the spooler was already open with attributes that conflict with those in the current open. This error is reported only for an open to standard output and only if one or more of the following are true:

- The spooling levels of the two opens are different.
- The new open specifies any level-2 arguments.

Effect. If the rejected request was initiated from a FORTRAN I/O statement that includes either an IOSTAT or ERR parameter, control is returned to the FORTRAN routine. Otherwise, the FORTRAN run-time library terminates the program.

Recovery. Coordinate how routines in your program use standard output.

H

Hollerith Constants and Punch Card Codes

Topics covered in this section include:

A Hollerith string defines a constant character string. Hollerith data was widely used to represent character strings prior to FORTRAN 77.

The form of a Hollerith constant is:

```
nHstring
```

n

is an unsigned, nonzero integer constant that specifies the number of characters in *string*.

string

is a string of *n* characters.

In HP FORTRAN, you can use Hollerith constants as:

- Actual parameters to subprograms
- Constant strings in FORMAT statements
- Input values that you read with a READ statement
- Initialization values in DATA statements

The following Hollerith constant specifies a string of four characters; the characters are NEWS:

```
4HNEWS
```

The following Hollerith constant specifies a string of eight characters; the characters are ++DATE++

```
8H++DATE++
```

Blank characters in a Hollerith constant are significant.

If you specify a Hollerith constant as an actual parameter in a CALL statement, the size of the Hollerith constant must be appropriate to the data type of the subprogram's

dummy argument. Table H-1 shows the relationship between the data type of a dummy argument and the maximum number of characters in a Hollerith string.

Table H-1. Hollerith Constant String Lengths

Entity	Maximum Characters	Example
INTEGER*2	2	DATA j /2H++/
LOGICAL	2	DATA c /2Hno/
INTEGER*4	4	DATA k /4Hplus/
REAL	4	DATA c /4Hwork/
LOGICAL*4	4	DATA a /4Htrue/
INTEGER*8	8	DATA m /8HPersonal/

The following example uses a DATA statement to initialize the quadrupleword integer K to the string “JANUARY ”. Because January has only seven letters in it, a single blank after the Y in January is included in the Hollerith constant:

```
INTEGER*8 k
DATA k/8HJANUARY /
```

The following example shows a CALL statement that specifies a Hollerith constant:

```
CALL sub(2HNO)
```

The following example uses a Hollerith constant in a FORMAT statement:

```
    WRITE (10, 100) balance
100 FORMAT(1x, 11HBalance is , F8.2)
```

If the length of string is less than the number of characters the entity can contain, the characters are left justified in the entity and the remaining character positions are filled with blanks.

Editing Hollerith Data

Use the A w edit descriptor with Hollerith data when an I/O list item is of type integer, real, or logical; for example:

```
    DATA j,a/54321,4H$$$$/
    WRITE (*,33) j,a
33 FORMAT (I5,A4)
```

Hollerith Constants as Subprogram Arguments

You can use a Hollerith constant as an actual argument in a subprogram reference. The corresponding dummy argument must be of type integer, real, or logical:

```
SUBROUTINE example(cost,date)
INTEGER*8 date
.
END
PROGRAM Main
CALL example(price,8H4-21-85 )
.
END
```

Hollerith Punch Card Codes

Table H-2 lists the Hollerith characters, their representations on a punch card (shown as rows punched in a given column), and their ASCII internal (octal) equivalents.

Table H-2. Hollerith Characters (page 1 of 4)

Hollerith Character	Hollerith Punch	ASCII Octal Equivalent
space	no punch	%040
&	12	%046
-	11	%055
0	0	%060
{	12,0	%173
|	12,11	%174
}	11,0	%175
1	1	%061
2	2	%062
3	3	%063
4	4	%064
5	5	%065
6	6	%066
7	7	%067
8	8	%068
9	9	%069
A	12,1	%101

Table H-2. Hollerith Characters (page 2 of 4)

Hollerith Character	Hollerith Punch	ASCII Octal Equivalent
B	12,2	%102
C	12,3	%103
D	12,4	%104
E	12,5	%105
F	12,6	%106
G	12,7	%107
H	12,8	%110
I	12,9	%111
J	11,1	%112
K	11,2	%113
L	11,3	%114
M	11,4	%115
N	11,5	%116
O	11,6	%117
P	11,7	%120
Q	11,8	%121
R	11,9	%122
/	0,1	%057
S	0,2	%123
T	0,3	%124
U	0,4	%125
V	0,5	%126
W	0,6	%127
X	0,7	%130
Y	0,8	%131
Z	0,9	%132
a	12,0,1	%141
b	12,0,2	%142
c	12,0,3	%143
d	12,0,4	%144
e	12,0,5	%145
f	12,0,6	%146
g	12,0,7	%147
h	12,0,8	%150

Table H-2. Hollerith Characters (page 3 of 4)

Hollerith Character	Hollerith Punch	ASCII Octal Equivalent
i	12,0,9	%151
j	12,11,1	%152
k	12,11,2	%153
l	12,11,3	%154
m	12,11,4	%155
n	12,11,5	%156
o	12,11,6	%157
p	12,11,7	%160
q	12,11,8	%161
r	12,11,9	%162
~	11,0,1	%176
s	11,0,2	%163
t	11,0,3	%164
u	11,0,4	%165
v	11,0,5	%166
w	11,0,6	%167
x	11,0,7	%170
y	11,0,8	%171
z	11,0,9	%172
o	8,1	%140
:	8,2	%072
#	8,3	%042
@	8,4	%100
‘	8,5	%047
=	8,6	%075
"	8,7	%042
[	12,8,2	%133
.	12,8,3	%056
<	12,8,4	%074
(	12,8,5	%050
+	12,8,6	%053
!	12,8,7	%041
]	11,8,2	%135
$	11,8,3	%044

Table H-2. Hollerith Characters (page 4 of 4)

Hollerith Character	Hollerith Punch	ASCII Octal Equivalent
*	11,8,4	%052
)	11,8,5	%051
;	11,8,6	%073
^	11,8,7	%136
\	0,8,2	%134
,	0,8,3	%054
%	0,8,4	%045
_	0,8,5	%137
>	0,8,6	%176
?	0,8,7	%077

Glossary

ASSIGN. An HP Tandem Advanced Command Language (TACL) command you can use to associate a file name with a logical file of a program or to assign a physical device to logical entities that an application uses.

BIND. A program invoked during system generation that creates TNS object (file code 100) system code files and system library files.

binding. The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled TNS object files to produce a target object file.

BINSERV. A version of the Binder program that is integrated with the C, COBOL85, FORTRAN, and TAL compilers.

block. A grouping of one or more system enclosures that an HP NonStop™ S-series system recognizes and supports as one unit. A block can consist of either one processor enclosure, one I/O enclosure, or one processor enclosure with one or more I/O enclosures attached.

checkpoint. The operation by which information in the primary process of a NonStop process pair is sent from the primary process to the backup process. *See also* stack checkpoint. and takeover point..

code block. The smallest independently relocatable piece of a program. Code blocks contain executable machine instructions and possibly inline constant data. Compare with data block.

collector. *See* spooler collector.

COMMON block. A data block whose contents can be referenced by all modules.

Common Run-Time Environment (CRE). A set of services, implemented by the CRE library, that supports mixed-language programs on D-series systems. Contrast with language-specific run-time environment.

Common Run-Time Environment (CRE) library. A collection of routines that supports requests for services managed by the CRE, such as I/O and heap management, math and string functions, exception handling, and error reporting. CRE library routines are used by D-series C, COBOL85, FORTRAN, and Pascal run-time libraries.

compilation unit. The object code produced by a single run of a compiler.

compiler directive. A compiler option with which you control compilation, compiler listings, and object code generation. For example, compiler directives enable you to compile parts of a source file conditionally or to suppress parts of a compiler listing.

connection. With respect to the CRE, a connection is a path managed by the CRE from a process to a Guardian file. Each connection is a unique path to the same Guardian file

and to the same open of that file. The CRE manages the connection. The CRE provides connection services for shared files.

In FORTRAN, a connection is an association between a unit number and a file.

CRE. *See* Common Run-Time Environment (CRE).

Crossref. A stand-alone product that collects and prints cross-reference information for your program.

C-series system. A system that is running a C-series version of the operating system.

data block. The smallest independently relocatable piece of a program. Data blocks contain statically allocated variables or constants. Compare with code block.

data segment. A virtual memory segment holding data. Every process begins with its own data segments for program global variables and runtime stacks (and for some libraries, instance data). Additional data segments can be dynamically created.

DEFINE. An HP Tandem Advanced Command Language (TACL) command you can use to specify a named set of attributes and values to pass to a process.

D-series system. A system that is running a D-series version of the operating system.

entry point. A location where a code block can be accessed.

extended data segment. A segment that provides up to 127.5 megabytes of data storage. A process can have more than one extended data segment.

file ID. The last of the four parts of a file name; the first three parts are node name (system name), volume name, and subvolume name.

file name. A sequence of four names, separated by periods, that specifies the location of a file. A file name consists of a:

- Node name (system name)
- Volume name
- Subvolume name
- File ID

global data. The identifiers in a data block that is accessible to all procedures in a program.

high PIN. A process identification number (PIN) that is greater than 255. Contrast with low PIN.

home terminal. Usually the terminal from which a process is started.

level-1 spooling. A method of spooling files in a HP NonStop environment. With level-1 spooling, a file uses default spooling parameters. The program writes records to a spooler collector by calling standard file system procedures such as WRITE.

level-2 spooling. A method of spooling files in a HP NonStop environment. With level-2 spooling, the program specifies spooling parameters for the file and writes records to a spooler collector by using standard file system procedures such as WRITE.

level-3 spooling. A method of spooling files in a HP NonStop environment. With level-3 spooling the program specifies spooling parameters, spooler data is buffered, and the program writes records to a spooler collector by using spooler interface procedures. For more information, see the *Spooler Programmer's Guide*.

local data. Data that you declare within a procedure.

low PIN. A process identification number (PIN) in the range 0 through 254. Contrast with high PIN.

lower 32K-word area. The lower half of the user data segment.

main routine. The first routine to execute when a program is run. The main routine determines the run-time environment for a program. In FORTRAN, a routine that does not begin with a SUBROUTINE or FUNCTION statement is a main routine. A main routine can optionally begin with a PROGRAM statement. You can use a PROGRAM statement to assign a name to the main program.

mixed-language program. A program that contains routines written in different HP-defined programming languages.

node name. A D-series term that identifies the name of a set of processors on a network. A node name always begins with a backslash character. A node name in a D-series system serves the same purpose as a system name in a C-series system.

object file. A file generated by a compiler or binder that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. An object file might be a complete program that is ready for immediate execution, or it might require binding with other object files before execution.

own data block. A data block that contains the data declared in a FORTRAN subprogram that executes a SAVE statement.

PARAM command. A TACL command with which you associate an ASCII value with a parameter name.

PIN. *See* process identification number (PIN).

primary data space. The area of the user data segment in which pointers and directly addressed variables are located.

process. A program that has been submitted to the operating system for execution, or a program that is currently running in the computer.

process identification number (PIN). A number that uniquely identifies a process running in a processor. The same number can exist in other processors in the same system. Internally, a PIN is used as an index into the process control block (PCB) table.

program file. An executable object file. It must contain an entry point with the MAIN attribute.

run-time environment. The run-time services provided to a program by library routines.

run-time library. A collection of routines that supports requests for services such as I/O and heap management, math and string functions, exception handling, and error reporting.

single-language program. A program in which all routines are written in the same programming language.

spooler collector. A process to which applications write data that is to be written to a printer device.

stack checkpoint. A checkpoint that sends a copy of all current local and sublocal data to a backup process by including the clause STACK = 'YES' in a CHECKPOINT, OPEN, or CLOSE statement. I/O statements such as READ and WRITE statements do stack checkpoints if they implicitly open a unit.

standard file. A file that your program can use with minimal or no changes to the file's attributes. The CRE supports three standard files—standard input, standard output, and standard log—that correspond to the files STDIN, STDOUT, and STDERR in a C programming environment.

standard input. A file from which a program can read sequential records. Each program defines how standard input is used according to the needs of the application. Standard input is analogous to the file STDIN in C. If you run your program from a TACL command line, standard input corresponds to the file you specify with the IN run-option.

standard log. A file to which a program can write sequential records. The records written to standard log are usually informational, warning, or error messages that describe exceptional conditions in a program. Standard log is analogous to the file STDERR in C.

standard output. A file to which a program can write sequential records. The program defines how standard output is used according to the needs of the application. Standard output is analogous to the file STDOUT in C. If you run your program from a TACL command line, standard output corresponds to the file you specify with the OUT run-option.

sublocal data. In FORTRAN, dummy arguments to statement functions.

system. All the processors, controllers, firmware, peripheral devices, software, and related components that are directly connected together to form an entity that is managed by one HP NonStop™ Kernel operating system image and operated as one computer.

system name. A C-series term that identifies a system on a network. A system name always begins with a backslash character. A system name in a C-series system serves the same purpose as a node name in a D-series system.

system procedure. A procedure supplied by the operating system.

takeover point. Location of the FORTRAN instruction that immediately follows the last stack CHECKPOINT statement. The FORTRAN run-time library implicitly executes a stack checkpoint when a program executes an OPEN or CLOSE statement that specifies the STACK = 'YES' option specifier.

TNS. HP computers that support the HP NonStop™ Kernel operating system and that are based on complex instruction-set computing (CISC) technology. TNS processors implement the TNS instruction set. Contrast with TNS/R.

TNS/R. HP computers that support the HP NonStop™ Kernel operating system and that are based on reduced instruction-set computing (RISC) technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture. Systems with these processors include most of the HP NonStop™ servers. Contrast with TNS.

target file. The output object file produced by the Binder.

TNS object code. The TNS instructions that result from processing program source code with a TNS language compiler. TNS object code executes on TNS and TNS/R systems.

TNS object file. The object file created by a TNS compiler. The file contains TNS instructions and other information needed to construct the code spaces and the initial data for a TNS process.

upper 32K-word area. The upper half of the user data segment.

user data segment. A data segment in which your program stores global and local data and a run-time stack; and in which the FORTRAN run-time library stores its data.

user library. A logically distinct part of the HP NonStop™ Kernel operating system that consists of procedures that the operating system can link to a program file at run time.

user library.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z

Special Characters