August 1978

This manual describes how to compile, link, debug, and execute programs written in the VAX-11 FORTRAN IV-PLUS language, using the facilities of the VAX/VMS operating system. It also contains other information of interest to FORTRAN programmers, such as: error processing, programming efficiency, compatibility with PDP-11 FORTRAN, and FORTRAN input/output.

VAX-11 FORTRAN IV-PLUS
User's Guide

Order No. AA-D035A-TE

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V01
SOFTWARE VERSION:	VAX-11 FORTRAN IV-PLUS V01

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1978

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

CONTENTS

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

The VAX-11 FORTRAN IV-PLUS User's Guide is intended for use in developing new FORTRAN programs, and compiling and executing existing FORTRAN programs on VAX/VMS systems. FORTRAN IV-PLUS language elements supported on VAX-11 are described in the VAX-11 FORTRAN IV-PLUS Language Reference Manual.

INTENDED AUDIENCE

This manual is designed for programmers who have a working knowledge of FORTRAN. Detailed knowledge of VAX/VMS is helpful but not essential; familiarity with the VAX/VMS Primer is recommended. Some sections of this book, however, (condition handling, for instance) require more extensive understanding of the operating system. In such sections, you are directed to the appropriate manual(s) for the required additional information.

STRUCTURE OF THIS DOCUMENT

This manual is organized as follows:

- Chapter 1 contains the information needed to compile, link, and execute a FORTRAN program.
- Chapter 2 covers the debugging process; use of the VAX-11 Symbolic Debugger is described.
- Chapter 3 provides information about FORTRAN input/output, including details on the use of logical names, file conventions, record structure, and use of certain OPEN statement keywords.
- Chapter 4 discusses character data, and includes examples of how character data can be manipulated.
- Chapter 5 discusses the conventions followed in calling procedures, especially the argument-passing conventions.
- Chapter 6 describes error processing; in particular, the condition handling facility and how to use it. This chapter is intended for users with in-depth knowledge of VAX/VMS.
- Chapter 7 describes the relationship between VAX-11 FORTRAN IV-PLUS and the VAX-11 system, with particular emphasis on program section usage, data types, functions, DO loops, and floating point data representation.

- Chapter 8 covers programming considerations relevant to typical FORTRAN applications.

- Appendixes A through D summarize internal data representation, diagnostic messages, system-supplied functions, and compatibility between VAX-11 FORTRAN and PDP-11 FORTRAN.

ASSOCIATED DOCUMENTS

The following documents are relevant to VAX-11 FORTRAN IV-PLUS programming:

- VAX/VMS Primer
- VAX-11 FORTRAN IV-PLUS Language Reference Manual
- VAX/VMS Command Language User's Guide
- VAX-11 Common Run-Time Procedure Library Reference Manual
- VAX-11 Linker Reference Manual
- VAX-11 Symbolic Debugger Reference Manual
- VAX/VMS System Services Reference Manual
- VAX-11/780 Architecture Handbook

For a complete list of VAX-11 software documents, see the VAX-11 Information Directory.

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are observed in this manual, as in the other VAX-11 documents:

- Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown

- Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice

- Brackets ([ ]) indicate optional elements

- Braces ({ }) are used to enclose lists from which one element is to be chosen

- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times

CHAPTER 1

USING VAX-11 FORTRAN IV-PLUS

VAX-11 FORTRAN IV-PLUS is based on American National Standard FORTRAN X3.9-1966. It is also a compatible superset of PDP-11 FORTRAN IV-PLUS. VAX-11 FORTRAN IV-PLUS provides the following extensions:

- Symbolic names up to 15 characters, including the underline and dollar sign characters
- FORTRAN 77 character data type
- FORTRAN 77 block IF constructs
- Relative file organization
- Standard CALL facility
- Hexadecimal constants and field descriptors
- Symbolic debugging facility
- Increased file manipulation facilities

Because VAX-11 FORTRAN IV-PLUS is a compatible superset of PDP-11 FORTRAN, you can execute existing PDP-11 FORTRAN programs on VAX-11 hardware. (Note that throughout the rest of this manual, unless explicitly stated otherwise, VAX-11 FORTRAN IV-PLUS will usually be referred to simply as FORTRAN.)

1.1 CREATING AND EXECUTING A PROGRAM

Figure 1-1 shows how a program proceeds from inception to execution. You specify the steps shown in Figure 1-1 by entering commands to the VAX-11 system. As shown, the commands are:

```
$ EDIT file-spec
$ FORTRAN file-spec
$ LINK file-spec
$ RUN file-spec
```

With each command, you include information that further defines what you want the system to do. Of prime importance is the file specification, indicating the file to be processed. You can also specify qualifiers that modify the processing performed by the system.

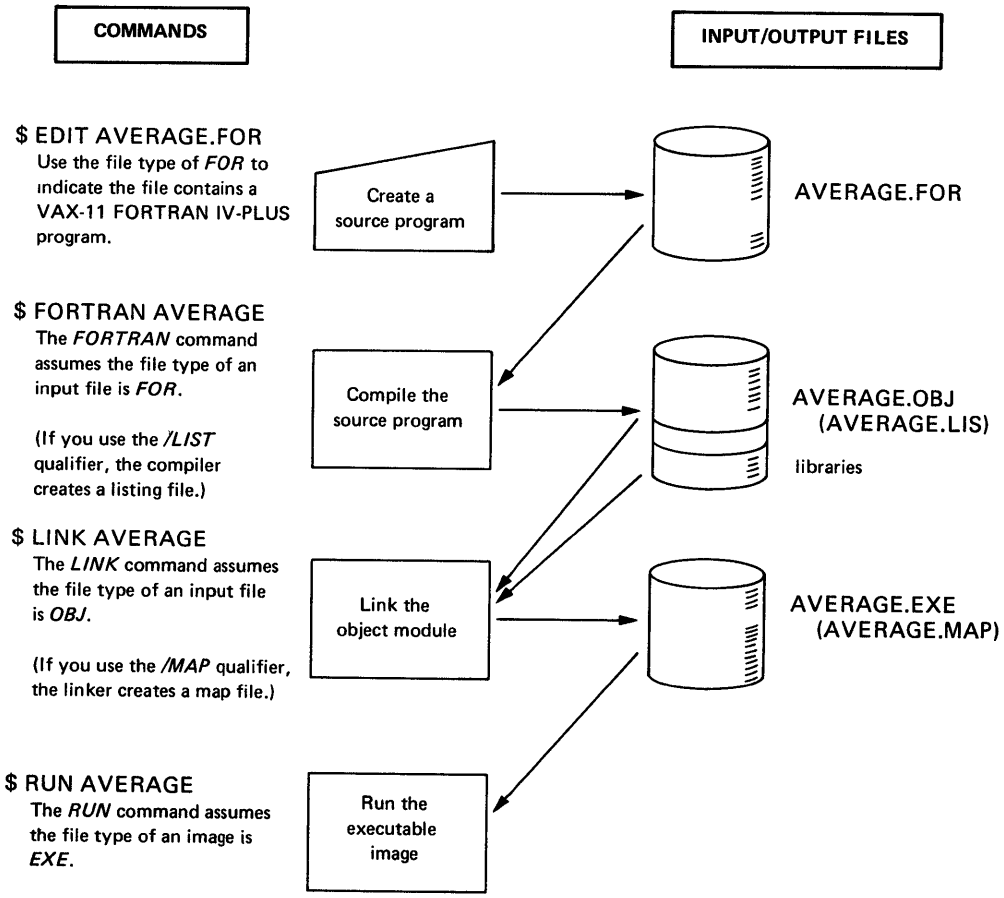


Figure 1-1 Program Development Process

1.1.1 File Specifications

A file specification indicates the input file to be processed, or the output file to be produced. File specifications have the form:

```
node::device:[directory]filename.filetype.version
```

node
Specifies a network node name. This is applicable only to systems that support VAX-11 DECnet.

device
Identifies the device on which a file is stored or is to be written.

directory
Identifies the name of the directory under which the file is cataloged, on the device specified. (You can delimit the directory name with either square brackets, as shown, or angle brackets < >).

filename
Identifies the file by its name; filename can be up to 9 characters long.

filetype
Describes the kind of data in the file; filetype can be up to 3 characters long.

version
Defines which version of the file is desired. Versions are identified by a decimal number, which is incremented by 1 each time a new version of a file is created. Either a semicolon or a period can be used to separate filetype and version.

You need not explicitly state all elements of a file specification each time you compile, link, or execute a program. The only part of the file specification that is usually required is the file name. If you omit any other part of the file specification, a default value is used. Table 1-1 summarizes the default values.

Table 1-1
File Specification Defaults

Optional Element	Default Value
node	Local network node
device	User's current default device
directory	User's current default directory
filetype	Depends on usage: Input to compiler FOR Output from compiler OBJ Input to linker OBJ Output from linker EXE Input to RUN command EXE Compiler source listing LIS Linker map listing MAP Input to executing program DAT Output from executing program DAT
version	Input: highest existing version Output: highest existing version, plus 1

If you request compilation of a FORTRAN program, and you specify only a file name, the compiler can process the source program if it finds a file with the specified file name that:

- Is stored on the default device
- Is cataloged under the default directory name
- Has a file type of FOR

If more than one file meets these conditions, the compiler chooses the one with the highest version number.

For example, assume that your default device is DBA0, your default directory is SMITH, and you supply the following file specification to the compiler:

```
CIRCLE
```

The compiler will search device DBA0, in directory SMITH, seeking the highest version of CIRCLE.FOR. If you do not specify an output file, the compiler will generate the file CIRCLE.OBJ, store it on device DBA0 in directory SMITH, and assign it a version number 1 higher than any other version of CIRCLE.OBJ currently cataloged in directory SMITH on DBA0.

1.1.2 Qualifiers

Qualifiers specify special actions to be performed, and can be either command qualifiers or file qualifiers. Qualifiers have the form:

```
/qualifier
```

Many qualifiers have a corresponding negative form that negates the action that the qualifier specifies. The negative form is /NOqualifier. For example, the qualifier /LIST tells the compiler to produce a listing file; /NOLIST tells the compiler not to produce a listing file.

Defaults have been established for each qualifier, based on the actions that are appropriate in most cases. Sections 1.2.2 and 1.3, which describe each command's qualifiers, contain tables indicating the defaults.

You can specify qualifiers so that either all files included in the command are affected, or only certain files are affected. If the qualifier immediately follows the command name, it applies to all files.

For example:

```
$ FORTRAN/LIST ABC,XYZ,RST
```

If you specify the above, you will receive listing files for ABC, XYZ, and RST.

If you include a qualifier as part of a file specification, it will (with certain exceptions) affect only the file with which it is associated. For example:

```
$ FORTRAN/LIST ABC,XYZ/NOLIST,RST
```

As a result of this command, listing files are created for ABC and RST, but not for XYZ.

Qualifiers included with file specifications that are part of a concatenated list of input files are exceptions to this rule. See Example 5 in Section 1.2.1, below.

1.2 COMPILATION

To compile a source program, use the FORTRAN command. The format of the FORTRAN command is:

```
$ FORTRAN[/qualifiers]  file-spec-list[/qualifiers]
```

/qualifiers
Codes indicating special actions to be performed by the compiler.

file-spec-list
Specification of the source file(s) containing the program to be compiled. You can specify more than one source file. If source file specifications are separated by commas, the programs are compiled separately. If source file specifications are separated by plus signs, the files are concatenated and compiled as one program.

In interactive mode, you can also enter the file specification on a separate line by typing a carriage return after $ FORTRAN. The system responds with the prompt:

```
$_File:
```

Type the file specification immediately after the $_File: prompt.

1.2.1 Specifying Output Files

The output produced by the compiler includes object files and listing files. You can control the production of these files by using the appropriate qualifiers with the FORTRAN command. If you do not specify otherwise, the compiler generates an object file. In interactive mode, the compiler generates no listing file, by default. In interactive mode, you must use the /LIST qualifier to generate a listing file. In batch mode, however, just the opposite is true: by default, the compiler will produce a listing file. To suppress the listing file, you must specify the /NOLIST qualifier.

During the early stages of program development, you may find it helpful to suppress the production of object files until your source program compiles without errors. Use the /NOOBJECT qualifier. If you do not specify /NOOBJECT, the compiler generates object files as follows:

- If you specify one source file, one object file is generated
- If you specify multiple source files, separated by plus signs, the source files are concatenated and compiled, and one object file is generated
- If you specify multiple source files, separated by commas, each source file is compiled separately, and an object file is generated for each source file
- You can use both plus signs and commas in the same command line to produce different combinations of concatenated and separate object files (see Example 4 below)

To produce an object file with an explicit file specification, you must use the /OBJECT qualifier, in the form /OBJECT=file-spec (see Section 1.2.2.8). Otherwise, the object file will have the name of its corresponding source file, and a file type of OBJ. By default,

the object file produced from concatenated source files has the name of the first source file. All other file specification attributes (node, device, directory, and version) will assume the default attributes.

Examples:

1. $ FORTRAN/LIST AAA,BBB,CCC

   Source files AAA.FOR, BBB.FOR, and CCC.FOR are compiled as separate files, producing object files named AAA.OBJ, BBB.OBJ, and CCC.OBJ; and listing files named AAA.LIS, BBB.LIS, and CCC.LIS.

2. $ FORTRAN XXX+YYY+ZZZ

   Source files XXX.FOR, YYY.FOR, and ZZZ.FOR are concatenated and compiled as one file, producing an object file named XXX.OBJ.

3. $ FORTRAN/OBJECT=SQUARE
   $_FILE: CIRCLE

   The source file CIRCLE.FOR is compiled, producing an object file named SQUARE.OBJ, but no listing file. (This example applies to interactive mode only.)

4. $ FORTRAN AAA+BBB,CCC/LIST

   Two object files are produced: AAA.OBJ (comprising AAA.FOR and BBB.FOR), and CCC.OBJ (comprising CCC.FOR). One listing file is produced: CCC.LIS.

5. $ FORTRAN ABC+CIRC/NOOBJECT+XYZ

   When you include a qualifier in a list of files that are to be concatenated, the qualifier affects all files in the list. Thus, in the command shown, you will completely suppress the object file. That is, source files ABC.FOR, CIRC.FOR, and XYZ.FOR will be concatenated and compiled, but no object file will be produced.

1.2.2 FORTRAN Qualifiers

In many cases, the simplest form of the FORTRAN command is sufficient for file processing. In some cases, however, you will need to use the FORTRAN qualifiers that specify special processing.

A FORTRAN qualifier has the form:

```
/aa[=y]
```

where aa is the qualifier's name, and y represents a qualifier value. Note that many qualifiers accept no value; the purpose of these qualifiers is simply to activate or deactivate a particular form of processing.

To specify a list of qualifier values, enclose them in parentheses. For example:

```
/CHECK=(BOUNDS,OVERFLOW)
```

Table 1-2 lists the qualifiers you can use with the FORTRAN command. Sections 1.2.2.1 through 1.2.2.11 describe each qualifier in detail.

Table 1-2
FORTRAN Command Qualifiers

Qualifier	Negative Form	Default
/CHECK=([NO]BOUNDS, [NO]OVERFLOW, ALL, NONE)	/NOCHECK	/CHECK=OVERFLOW
/CONTINUATIONS=n	None	/CONTINUATIONS=19
/DEBUG=([NO]SYMBOLS, [NO]TRACEBACK, ALL, NONE)	/NODEBUG	/DEBUG=TRACEBACK
/D_LINES	/NOD_LINES	/NOD_LINES
/I4	/NOI4	/I4
/LIST[=file-spec]	/NOLIST	/NOLIST (interactive) /LIST (batch)
/MACHINE_CODE	/NOMACHINE_CODE	/NOMACHINE_CODE
/OBJECT[=file-spec]	/NOOBJECT	/OBJECT
/OPTIMIZE	/NOOPTIMIZE	/OPTIMIZE
/WARNINGS	/NOWARNINGS	/WARNINGS
/WORK_FILES=n	None	/WORK_FILES=2

1.2.2.1 **CHECK Qualifier** - At run time, this qualifier causes the compiler to produce code to check your program for the conditions indicated. It has the form:

```
/CHECK =([NO]BOUNDS
         [NO]OVERFLOW
         ALL
         NONE)
```

BOUNDS
Array references are checked to ensure that they are within the array address boundaries specified. Note, however, that array bound checking is not performed for arrays that are dummy arguments, and for which the last dimension bound is specified as 1. For example:

```
DIMENSION A(1)
```

OVERFLOW
BYTE, INTEGER*2, and INTEGER*4 calculations are checked for arithmetic overflow.

ALL
Both OVERFLOW and BOUNDS checks are performed.

NONE
Neither check is performed.

The default is /CHECK=OVERFLOW. Note that /CHECK is the equivalent of /CHECK=ALL, and /NOCHECK is the equivalent of /CHECK=NONE.

If you specify /CHECK=BOUNDS or /CHECK=OVERFLOW, the other check is implicitly canceled.

1.2.2.2 **CONTINUATIONS Qualifier** - This qualifier specifies the number of continuation lines allowed in the source program. It has the form:

/CONTINUATIONS=n

You can specify a value from 0 to 99 for n. If you omit /CONTINUATIONS, the default value is 19.

1.2.2.3 **DEBUG Qualifier** - This qualifier specifies that the compiler is to provide information for use by the VAX-11 Symbolic Debugger and the run-time error traceback mechanism. It has the form:

```
/DEBUG = ( [NO]SYMBOLS   )
         ( [NO]TRACEBACK )
         ( ALL           )
         ( NONE          )
```

SYMBOLS
The compiler provides the debugger with local symbol definitions for user-defined variables, arrays (including dimension information), and labels of executable statements.

TRACEBACK
The compiler provides an address correlation table so the debugger and the run-time error traceback mechanism can translate absolute addresses into source program routine names and compiler-generated line numbers.

ALL
The compiler provides both local symbol definitions and an address correlation table.

NONE
The compiler provides no debugging information.

The default is /DEBUG=TRACEBACK. Note that /DEBUG is the equivalent of /DEBUG=ALL, and /NODEBUG is the equivalent of /DEBUG=NONE. If you specify either /DEBUG=TRACEBACK or /DEBUG=SYMBOLS, the other is implicitly canceled.

For more information on debugging and traceback, see Section 1.5 and Chapter 2.

1.2.2.4 **D_LINES Qualifier** - This qualifier specifies that lines with a D in column 1 are to be compiled. It has the form:

```
/D_LINES
```

The default is /NOD_LINES, which means that lines with a D in column 1 are treated as comments.

1.2.2.5 **I4 Qualifier** - This qualifier controls how the compiler interprets INTEGER and LOGICAL declarations for which a length is not specified. It has the form:

```
/I4,
```

The default is /I4, which causes the compiler to interpret INTEGER and LOGICAL declarations as INTEGER*4 and LOGICAL*4. If you specify /NOI4, the compiler interprets them as INTEGER*2 and LOGICAL*2.

1.2.2.6 **LIST Qualifier** - This qualifier produces a source listing file. It has the form:

```
/LIST[=file-spec]
```

You can include a file specification for the listing file. If you do not, it defaults to the name of the first source file, and a file type of LIS.

The compiler does not produce a listing file in interactive mode unless you include the /LIST qualifier. In batch mode, the compiler produces a listing file by default. In either case, the listing file is not automatically printed. You must use the PRINT command to obtain a line printer copy of the listing file.

See Section 1.7 for a sample listing.

1.2.2.7 **MACHINE_CODE Qualifier** - This qualifier specifies that the listing file is to include a listing of the object code generated by the compiler. It has the form:

```
/MACHINE_CODE
```

This qualifier is ignored if no listing file is being generated.

The default is /NOMACHINE_CODE.

1.2.2.8 **OBJECT Qualifier** - This qualifier can be used when you want to specify the name of the object file. It has the form:

```
/OBJECT[=file-spec]
```

The default is /OBJECT. The negative form, /NOOBJECT, can be used to suppress object code; for example, when you only want to test the source program for compilation errors.

If you omit the file specification, the object file defaults to the name of the first source file, and a file type of OBJ.

1.2.2.9 **OPTIMIZE Qualifier** - This qualifier tells the compiler to produce optimized code. It has the form:

```
/OPTIMIZE
```

The default is /OPTIMIZE. The negative form (/NOOPTIMIZE) should be used to ensure that the debugger has sufficient information to help you locate errors in your source program (see Section 2.9).

1.2.2.10 **WARNINGS Qualifier** - This qualifier specifies whether the compiler is to generate diagnostic messages in response to warning-level (W) errors. It has the form:

```
/WARNINGS
```

The compiler generates warning (W) diagnostic messages by default. A warning diagnostic message indicates that the compiler has detected acceptable but nonstandard syntax, or has performed some corrective action; in either case, unexpected results may occur. To suppress W diagnostic messages, specify the negative form of this qualifier (/NOWARNINGS). The default is /WARNINGS.

Appendix B discusses compiler diagnostic messages.

1.2.2.11 **WORK_FILES Qualifier** - This qualifier changes the number of work files used by the compiler. It has the form:

```
/WORK_FILES=n
```

The value specified for n can be 1, 2, or 3.

Note that while a value of 1 may increase the speed of compilation, it restricts the size of programs that can be compiled. A value of 3 allows larger programs to be compiled, but may slow compilation. The default is /WORK_FILES=2.

1.3 LINKING

Before a compiled program can be executed, you must link the object file to produce an executable image file. Linking resolves all references in the object code, and establishes absolute addresses for symbolic locations. To link an object module, issue the LINK command, in the following general form:

```
$ LINK[/command-qualifiers] file-spec[/file-qualifiers]...
```

/command-qualifiers
Specify output file options.

file-spec
Specifies the input object file to be linked.

/file-qualifiers
Specify input file options.

In interactive mode you can issue the LINK command with no accompanying file specification. The system responds with the prompt:

```
$_File:
```

The file specification must be typed on the same line as the prompt. If there are too many file specifications to fit on one line, you can continue the line by typing a hyphen (-) as the last character of the line, and continuing on the next line.

You can enter multiple file specifications separated from each other by commas or plus signs. When used with the LINK command, the comma has the same effect as the plus sign: no matter which is used, a single executable image is created from the input files specified. If no output file is specified, the linker produces an executable image with the same name as the first object module, and a file type of EXE. Table 1-3 lists the linker qualifiers of particular interest to FORTRAN users. See the VAX-11 Linker Reference Manual for details on the linker.

Table 1-3
Linker Qualifiers

Command Qualifiers	Negative Form	Default
/EXECUTE[=file-spec]	/NOEXECUTE	/EXECUTE
/SHAREABLE[=file-spec]	/NOSHAREABLE	None
/MAP[=file-spec]	/NOMAP	/NOMAP (interactive) /MAP(batch)
/BRIEF	None	Not applicable
/FULL	None	Not applicable
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
/DEBUG	/NODEBUG	/NODEBUG
/TRACEBACK	/NOTRACEBACK	/TRACEBACK
Input File Qualifiers		
/LIBRARY		
/INCLUDE=module-name(s)		

1.3.1 Linker Command Qualifiers

You can specify qualifiers for the LINK command to modify the output of the linker. You can also define whether the debugging or the traceback facility is to be included.

Linker output consists of an image file and, optionally, a map file. The following qualifiers control the image file generated by the linker:

```
/EXECUTE=file-spec
/NOEXECUTE
/SHAREABLE=file-spec
```

These qualifiers are described in Section 1.3.1.1.

Map file qualifiers include:

```
/MAP[=file-spec]
/BRIEF
/FULL
/CROSS_REFERENCE
```

These qualifiers are described in Section 1.3.1.2.

The debugger and traceback qualifiers are:

```
/DEBUG
/TRACEBACK
```

These qualifiers are described in Section 1.3.1.3.

1.3.1.1 **Image File Qualifiers** - Image file qualifiers include:

```
/EXECUTE
/SHAREABLE
```

If you do not specify an image file qualifier, the default is /EXECUTE; the linker produces an executable image. To suppress production of an image, specify the negative form, as:

```
/NOEXECUTE
```

For example:

```
$ LINK/NOEXECUTE CIRCLE
```

The file CIRCLE.OBJ is linked, but no image is generated. The /NOEXECUTE qualifier is useful if you want to verify the results of linking an object file, without actually producing the image.

To designate a file specification for an executable image, use the /EXECUTE qualifier in the form:

```
/EXECUTE=file-spec
```

For example:

```
$ LINK/EXECUTE=TEST CIRCLE
```

The file CIRCLE.OBJ is linked, and the executable image generated is named TEST.EXE.

A shareable image is one that can be used in a number of different applications. It may be a private image you use for your own applications, or it may be installed in the system by the system manager for use by all users. To create a shareable image, specify the /SHAREABLE qualifier. For example:

```
$ LINK/SHAREABLE CIRCLE
```

To include a shareable image as input to the linker, you must use an options file, and specify the /OPTIONS qualifier in the LINK command. Refer to the VAX-11 Linker Reference Manual for details.

If you specify /NOSHAREABLE, the effect is similar to /NOEXECUTE. The linker processes the object code and the input as though it were going to produce a shareable image, but in fact no image is generated.

1.3.1.2 **Map File Qualifiers** - The map file qualifiers tell the linker whether a map file is to be generated, and, if so, the information it is to include. Map file qualifiers include:

```
/MAP
/BRIEF
/FULL
/CROSS_REFERENCE
```

The map qualifiers are specified as follows:

```
/MAP[=file-spec] [{/BRIEF}] [/CROSS_REFERENCE]
                 [{/FULL }]
```

The linker uses defaults to generate or suppress a map file. In interactive mode, the default is to suppress the map; in batch mode, the default is to generate the map.

If no file specification is included with /MAP, the map file has the name of the first input file, and a file type of MAP. It is stored on the default device, in the default directory.

The qualifiers /BRIEF and /FULL define the amount of information included in the map file, as follows:

- /BRIEF produces a summary of the image's characteristics, and a list of contributing modules.
- /FULL produces a summary of the image's characteristics and a list of contributing modules (as produced by /BRIEF); plus a list of global symbols and values, in symbol name order; and a summary of characteristics of image sections in the linked image.

By default, if neither /BRIEF nor /FULL is specified, the map file contains a summary of the image's characteristics and a list of contributing modules (as produced by /BRIEF), plus a list of global symbols and values, in symbol name order.

The /CROSS_REFERENCE qualifier can be used with either the default or /FULL map qualifiers, to request cross reference information for global symbols. This cross reference information indicates the object

modules that define and/or refer to global symbols encountered during linking. The default is /NOCROSS_REFERENCE.

1.3.1.3 **Debugging and Traceback Qualifiers** - The /DEBUG qualifier indicates that the VAX/VMS debugger (see Chapter 2) is to be included in the executable image. The default is /NODEBUG.

When the /TRACEBACK qualifier is specified, error messages are accompanied by a symbolic traceback showing the sequence of calls that transferred control to the program unit in which the error occurred. If you specify /NOTRACEBACK, this information is not produced. The default is /TRACEBACK. If you specify /DEBUG, the traceback capability is automatically included, and the /TRACEBACK qualifier is ignored. Figure 1-2 illustrates a typical traceback list. (See Section 1.5.1.)

1.3.2 Linker Input File Qualifiers

File qualifiers affect the input file specification. Input files can be object files; shareable files previously linked; or library files.

1.3.2.1 **/LIBRARY Qualifier** - The /LIBRARY qualifier has the form:

```
/LIBRARY
```

This qualifier specifies that the input file is an object-module library that is to be searched to resolve undefined symbols referenced in other input modules. The default file type is OLB.

1.3.2.2 **/INCLUDE Qualifier** - The /INCLUDE qualifier has the form:

```
/INCLUDE=module-name(s)
```

The qualifier specifies that the input file is an object-module library, and that the modules named are the only modules in that library that are to be explicitly included as input. At least one module name is required. To specify more than one, enclose the module names in parentheses, and separate them with commas. The default file type is OLB. The /LIBRARY qualifier can also be used with the same file specification, to indicate that the same library is also to be searched for unresolved references.

1.4 EXECUTION

The RUN command initiates execution of your program. It has the form:

```
$ RUN[/DEBUG] file-spec
```

The file name must be specified; default values are applied if you omit optional elements of the file specification. The default file type is EXE. The /DEBUG qualifier allows you to use the debugger, even though you omitted this qualifier from the FORTRAN and LINK commands. See Section 1.5 for details.

1.5 FINDING AND CORRECTING ERRORS

Both the compiler and the Run-Time Library include facilities for detecting and reporting errors. VAX/VMS also provides the debugger, to help you locate and correct errors. In addition to the debugger, there is a traceback facility that can also be used to track down errors that occur during program execution.

1.5.1 Error-Related Command Qualifiers

At each step in compiling, linking, and executing your program, you can specify command qualifiers that affect how errors are processed. At compile time, you can use the /DEBUG qualifier to ensure that symbolic information is created for use by the debugger. At link time you can also specify the /DEBUG qualifier to make the symbolic information available to the debugger. The same qualifier can be specified with the RUN command, to invoke the debugger.

Table 1-4 summarizes the /DEBUG and /TRACEBACK qualifiers.

Table 1-4
/DEBUG and /TRACEBACK Qualifiers

Qualifier	Command	Effect	Default
/DEBUG	FORTRAN	The FORTRAN compiler creates symbolic data needed by the debugger.	/DEBUG= (NOSYMBOLS, TRACEBACK)
/DEBUG	LINK	Symbolic data created by FORTRAN compiler is passed to the debugger.	/NODEBUG
/TRACEBACK	LINK	Traceback information is passed to the debugger. Traceback will be produced.	/TRACEBACK
/DEBUG	RUN	Invokes the debugger. The DBG> prompt will be displayed. Not needed if $ LINK/DEBUG was specified.	
/NODEBUG	RUN	If /DEBUG was specified in the LINK command, RUN/NODEBUG suppresses the DBG> prompt.	

If you use none of these qualifiers at any point in the compile-link-execute sequence, and an execution error occurs, you will receive a traceback list by default. However, you will not be able to invoke the debugger.

To perform symbolic debugging, you must use the /DEBUG qualifier with both the FORTRAN command and the LINK command. It then becomes unnecessary to specify it with the RUN command. If you omit /DEBUG from either the FORTRAN or LINK command, you can use it with the RUN command, to invoke the debugger. However, any debugging you perform must then be done by specifying addresses in absolute form, rather than symbolically.

If you linked your program with the debugger, but wish to execute the program without intervention by the debugger, specify

```
RUN/NODEBUG program
```

If you specify LINK/NOTRACEBACK, you will receive no traceback in the event of error. An example of a source program and a traceback is shown in Figure 1-2.

```
  0001            I=1
  0002            CONTINUE
  0003            J=2
  0004            CONTINUE
  0005            K=3
  0006            CALL SUB1
  0007            CONTINUE
  0008            END

  0001            SUBROUTINE SUB1
  0002            I=1
  0003            J=2
  0004            CALL SUB2
  0005            END

  0001            SUBROUTINE SUB2
  0002            COMPLEX W
  0003            COMPLEX Z

  0004            DATA W/(0.,0.)/

  0005            Z = LOG(W)
  0006            END

%MTH-F-LOGZERNEG, logarithm of zero or negative value
  user PC 00000449
%TRACE-F-TRACEBACK, symbolic stack dump follows

  module name     routine name      line   relative PC    absolute PC

                                           0000074C       0000074C
                                           0000081C       0000081C
                  SUB2                 5   00000011       00000449
                  SUB1                 4   00000017       00000437
                  T1$MAIN              6   0000001B       0000041B
```

Figure 1-2 Traceback List

The traceback is interpreted as follows:

When the error condition is detected, you receive the appropriate message, followed by the traceback information. In this example, a message is displayed by the Run-Time Library, indicating the nature of the error, and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls. Note that values may be produced for relative and absolute PC, with no corresponding values for routine name and line. These PC values reflect procedure calls internal to the Run-Time Library.

Of particular interest to you are the values listed under "routine name" and "line", the first of which shows what routine or subprogram called the Run-Time Library, which subsequently reported the error. The value given for "line" corresponds to the compiler-generated line number in the source program listing (not to be confused with editor-generated line numbers). Using this information, you can usually isolate the error in a short time.

If you specify either LINK/DEBUG, or RUN/DEBUG, the debugger assumes control of execution. If an error occurs, you do not receive a traceback list. To display traceback information, you can use the debugger command SHOW CALLS, as described in Section 1.5.2.

1.5.2 SHOW CALLS Command

When an error occurs in a program that is executing under the control of the debugger, no traceback list is produced. To generate a traceback list, use the SHOW CALLS command, which has the form:

```
DBG>SHOW CALLS
```

1.6 SAMPLE TERMINAL SESSION

A typical dialog between you and the system might appear as follows:

```
(RET)
Username: SMITH (RET)
Password: (RET)  (Your password is not displayed)

     WELCOME TO VAX/VMS RELEASE 1

$ EDIT CIRCLE.FOR (RET)
Input:DBA2:[SMITH]CIRCLE.FOR
00100
     (enter source program)
*E (RET)  (terminate edit session and write file to disk)
[DBA2:[SMITH]CIRCLE.FOR;1]
$ FORTRAN/NOOPTIMIZE/LIST/DEBUG CIRCLE

$ LINK/DEBUG CIRCLE

$ RUN CIRCLE
```

1.7 COMPILER LISTING FORMAT

The listing produced by the compiler consists of two or three sections, as follows:

- Source listing section
- Machine code listing section (optional)
- Storage map section

Sections 1.7.1 through 1.7.3 describe the compiler listing sections in detail.

1.7.1 Source Listing Section

The source listing section shows the source program as it appears in the input file with the addition of sequential line numbers generated by the compiler. Figure 1-3 shows a sample source listing section.

Note that line numbers are generated only for statements that are compiled; comment lines are not numbered, nor are lines with D in column 1 unless you specified /D_LINES.

Compiler-generated line numbers appear in the left margin. You can use them for debugging by using the %LINE specification in debugger commands (see Chapter 2). If the editor you use to create the source file generates line numbers, these numbers will also appear in the source listing. In this case, the editor-generated line numbers appear in the left margin, and the compiler-generated line numbers are shifted to the right. The %LINE specification applies to the compiler-generated line numbers, not the editor-generated line numbers.

Compile-time and run-time error messages that contain line numbers refer to the compiler-generated line numbers in the source listing section. See Appendix B for a summary of error messages.

```
0001            SUBROUTINE RELAX2(EPS)

0002            PARAMETER M=40, N=60
0003            DIMENSION X(0:M,0:N)
0004            COMMON X

0005            LOGICAL DONE

0006    1       DONE = .TRUE.

0007            DO 10 J = 1,N-1
0008            DO 10 I = 1,M-1
0009                XNEW = ( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) ) / 4
0010                IF ( ABS(XNEW-X(I,J)) .GT. EPS ) DONE = .FALSE.
0011    10          X(I,J) = XNEW

0012            IF (.NOT. DONE) GO TO 1

0013            RETURN
0014            END
```

Figure 1-3 Source Listing Section

1.7.2 Machine Code Listing Section

The machine code listing section provides a symbolic representation of the compiler-generated object code. This representation is similar to a VAX-11 MACRO assembly listing for the generated code and data.

The machine code listing section is optional. To include it in the listing file, you must specify:

```
$ FORTRAN/LIST/MACHINE_CODE
```

Figure 1-4 shows a sample machine code listing section.

```
                .TITLE  RELAX2
                .IDENT  01

0000            .PSECT  $BLANK
0000    X:

0000            .PSECT  $CODE
0000    RELAX2::
0000            .WORD   ^M<IV,R5,R6,R7,R8,R9,R10,R11>
0002            MOVAL   $LOCAL, R11
                                                                        ; 0006
0009    .1:
0009            MNEGL   #1, DONE(R11)
                                                                        ; 0007
000C            MOVL    #1, R7
000F            MOVAL   $BLANK, R5
0016    L$IANE:
                                                                        ; 0008
0016            MOVL    #1, R9
0019            MULL3   #41, R7, R6
001D    L$IAGG:
                                                                        ; 0009
001D            ADDL3   R9, R6, R10
0021            ADDF3   X+4(R5)[R10], X-4(R5)[R10], R0
0029            ADDF2   X-164(R5)[R10], R0
002F            ADDF2   X+164(R5)[R10], R0
0035            MULF3   #^X3F80, R0, R8
                                                                        ; 0010
003D            SUBF3   X(R5)[R10], R8, R0
0042            BICW2   #^X8000, R0
0047            CMPF    R0, @EPS(AP)
004B            BLEQ    L$IAPI
004D            CLRL    DONE(R11)
004F    L$IAPI:
                                                                        ; 0011
004F            MOVF    R8, X(R5)[R10]
0053            AOBLEQ  #39, R9, L$IAGG
0057            AOBLEQ  #59, R7, L$IANE
005B            MOVL    R7, J(R11)
005F            MOVF    R8, XNEW(R11)
0063            MOVL    R9, I(R11)
                                                                        ; 0012
0067            BLBC    DONE(R11), .1
006A            RET
                .END
```

Figure 1-4 Machine Code Listing Section

The first line of the machine code listing contains a .TITLE assembler directive, indicating the program unit to which the machine code corresponds. For a main program, the title is either as declared in a PROGRAM statement, or filename$MAIN, if you did not specify a PROGRAM statement. For subprograms, the title is the name of the subroutine or function. For a BLOCK DATA subprogram, the title is either the name declared in the BLOCK DATA statement, or filename$DATA, if you did not specify a name in the BLOCK DATA statement.

The lines following .TITLE provide information such as the contents of storage initialized for FORMAT statements, DATA statements, constants, and subprogram argument call lists. Machine instructions are represented by VAX-11 MACRO mnemonics and syntax. Compiler-generated line numbers corresponding to generated code lines are listed at the right margin before the machine code generated for the line.

The VAX-11 general registers (0 through 12) are represented by R0 through R12. When register 12 is used as the argument pointer, it is represented by AP; the frame pointer (register 13) is FP; the stack pointer (register 14) is SP, and the program counter (register 15) is

PC. Note that the relative PC for each instruction or data item is listed at the left margin, in hexadecimal.

Variables and arrays defined in the source program are shown in the machine code listing as they were defined in the source. Offsets from variables and arrays are shown in decimal.

FORTRAN source labels referred to in the source program are shown in the machine code listing with a dot (.) prefix. For example, if the source program refers to label 300, the label appears in the machine code listing as .300. Labels that appear in the source program, but that are not referred to or are deleted during compiler optimization, are ignored and do not appear in the machine code listing, unless you specified /NOOPTIMIZE.

The compiler may generate labels for its own use. These labels appear as L$xxxx, where the value of xxxx is unique for each such label in a program unit.

Integer constants are shown as signed integer values; real, double precision, and complex constants are shown as unsigned hexadecimal values preceded by ^X.

Addresses are represented by the program section name plus the hexadecimal offset within that program section. Changes from one program section to another are indicated by PSECT lines.

1.7.3 Storage Map Section

The storage map section of the compiler listing summarizes the following information:

- Program sections
- Entry points
- Statement functions
- Variables
- Arrays
- Labels
- Functions and subroutines
- Total memory allocated

Figure 1-5 shows a sample storage map section.

A summary section heading is not printed if no entries were generated for that section.

```
PROGRAM SECTIONS

    Name                    Bytes   Attributes

  0 $CODE                     107   PIC CON REL LCL   SHR   EXE   RD NOWRT LONG
  2 $LOCAL                     16   PIC CON REL LCL NOSHR NOEXE   RD   WRT LONG
  3 $BLANK                  10004   PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG

ENTRY POINTS

    Address  Type Name

  0-00000000       RELAX2

VARIABLES

    Address  Type Name                Address  Type Name                Address  Type Name

  2-00000000  L*4 DONE             AP-00000004@ L*4 EPS              2-00000008  I*4 I
  2-00000004  I*4 J                 2-0000000C  R*4 XNEW

ARRAYS

    Address  Type Name                    Bytes  Dimensions

  3-00000000  R*4 X                       10004  (0:40,0:60)

LABELS

    Address   Label       Address   Label

  0-00000009  1             **      10

Total Space Allocated = 10127 Bytes

COMPILER OPTIONS

/CHECK=(NOBOUNDS,OVERFLOW)
/DEBUG=(NOSYMBOLS,TRACEBACK)
/OPTIMIZE  /WARNINGS  /I4  /NOD_LINES
```

Figure 1-5 Storage Map Listing

Sizes are printed as a number of bytes, expressed in decimal. Data addresses are specified as an offset from the start of a program section, expressed in hexadecimal. The symbol AP can appear instead of a program section; in this case, the address refers to a dummy argument, expressed as the offset from the argument pointer (AP). Indirection is indicated by an at sign (@) following an address field. In this case, the address specified by the program section (or AP) plus the offset points to the address of the data, not to the data itself.

The program section summary describes each program section (PSECT) generated by the compiler. The descriptions include:

- PSECT number (used by most of the other summaries)
- Name
- Size
- Attributes

Chapter 7 describes PSECT usage and attributes.

The entry point summary lists all entry points and their addresses. If the program unit is a function, the declared data type of the entry point is also included.

The statement function summary lists the entry point address and data type of each statement function.

The variable summary lists all simple variables, with the data type and address of each.

The array summary is similar to the variable summary. In addition to data type and address, it gives the total array size and dimensions. If the array is an adjustable array, the size is shown as double asterisks (**), and each adjustable dimension bound is shown as a single asterisk (*).

The label summary lists all user-defined statement labels. FORMAT statement labels are suffixed with an apostrophe. If the label address field contains double asterisks (**), then the label was not used or referred to by the compiled code.

The functions and subroutines summary lists all external routine references made by the source program.

Following the summaries, the compiler prints the total memory allocated for all program sections compiled, in the form:

```
Total Space Allocated = nnn Bytes
```

1.7.4 Other Listing Information

The final entries on the compiler listing are the compiler qualifiers in effect for that compilation. For example:

```
COMPILER OPTIONS

/CHECK=(NOBOUNDS,OVERFLOW)
/DEBUG=(NOSYMBOLS,TRACEBACK)
/OPTIMIZE  /WARNINGS  /I4  /NOD_LINES
```

CHAPTER 2

DEBUGGING FORTRAN PROGRAMS

Debugging is the process of finding and correcting errors in executable programs; that is, in programs that have been compiled and linked without diagnostic messages, but that produce invalid results.

This chapter shows you how to use the VAX-11 Symbolic Debugger to debug FORTRAN programs.

2.1 OVERVIEW OF THE VAX-11 SYMBOLIC DEBUGGER

The VAX-11 Symbolic Debugger Reference Manual describes the VAX-11 debugger in detail. This section provides an overview of the debugger, showing a sample debugging session and introducing the debugger command syntax and symbol table.

2.1.1 Sample Debugging Terminal Session

Figure 2-1 illustrates a program that requires debugging. The program has been compiled and linked without diagnostic messages from either the compiler or the linker. (Appendix B summarizes compiler diagnostic messages.) However, the program produces erroneous results because of the missing asterisk in the exponentiation operator (RADIUS*2 should be RADIUS**2). This error is so obvious that you hardly need the services of the debugger to find it. However, for purposes of illustration, this example will deal with the error as though it were shrouded in obscurity.

```
          C       PROGRAM TO FIND THE AREA
          C       OF A CIRCLE

0001              PROGRAM CIRCLE
0002              TYPE 5
0003      5       FORMAT (' enter radius value ')
0004              ACCEPT 10,RADIUS
0005      10      FORMAT (F6.2)
0006              PI = 3.1415927
0007              AREA = PI*RADIUS*2
0008              TYPE 15,AREA
0009      15      FORMAT (' area of circle equals ',F10.3)
0010              STOP
0011              END
```

Figure 2-1 Sample FORTRAN Program: CIRCLE

The key to debugging is to find out what happens at critical points in your program. To do this, you need a way to stop execution at selected locations, and look at the contents of these locations to see if they contain the correct values. Points at which execution is stopped are called breakpoints. The SET BREAK command lets you specify where you want to stop the program.

To look at the contents of a location, use the EXAMINE command. To resume execution, use either the GO or STEP command. All DEBUG commands relevant to FORTRAN are discussed in subsequent sections of this chapter.

Figure 2-2 is an example of typical terminal dialog for a debugging session. The circled numbers (for example, (1)) are keyed to notes that follow the figure and explain the dialog.

```
$ FORTRAN/LIST/NOOPTIMIZE/DEBUG  CIRCLE  (1)
$ LINK/DEBUG  CIRCLE  (2)
$ RUN CIRCLE

  DEBUG Version 0.5-1   28 April 1978

%DEBUG-I-INITIAL, language is FORTRAN, scope and module set to CIRCLE  (3)
DBG>SET BREAK %LINE 7  (4)
DBG>GO  (5)
routine start pc is CIRCLE\CIRCLE
enter radius value
24.
break at pc = CIRCLE\CIRCLE %line 7  (6)
DBG>EXAMINE PI  (7)
CIRCLE\PI:      3.141593
DBG>EXAMINE RADIUS  (8)
CIRCLE\RADIUS:     24.00000
DBG>EXAMINE AREA  (9)
CIRCLE\AREA:     0.0000000
DBG>GO  (10)
start pc is CIRCLE\CIRCLE %line 7
area of circle equals     150.796
FORTRAN STOP
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'  (11)
DBG>EXIT  (12)
$
```

Figure 2-2 Sample Debugging Terminal Dialog

(1) Invoke the FORTRAN compiler, specifying the qualifiers shown. You should include the /NOOPTIMIZE qualifier if you intend to use symbolic debugging (see Section 2.9).

(2) Link your program, including the debugger.

(3) In response to the RUN command, the debugger displays its identification, indicating that your program will be executed under the debugger's control. Following the identification message, the debugger displays an INITIAL message, telling you the mode settings it has assumed for the language, scope, and module. The debugger derives the mode settings from the first module specified in the LINK command. If this message does not appear, or if the settings assumed are not appropriate, use the SET LANGUAGE, SET SCOPE, and SET MODULE commands, as described in Sections 2.2.1, 2.2.2, and 2.2.3.

❹ Place a breakpoint at an appropriate point in the program. This point should be one at which you will be able to examine key variables. Note: Breakpoints suspend execution just before the point specified.

❺ Begin program execution. The debugger displays the line number at which execution starts.

❻ The debugger announces that it has suspended execution at the specified breakpoint.

❼ Check the variable PI to make sure the correct value is stored there. The debugger displays the contents of PI, showing that its scope is in module CIRCLE.

❽ Check the variable RADIUS. The debugger shows that the specified value has been properly stored.

❾ Examine the variable AREA to make sure its contents are zero.

❿ Resume execution. The debugger displays a message indicating the point of program resumption.

⓫ Successful completion of the program is indicated by this message. However, as you can see, the result is incorrect.

⓬ Exit from the debugger.

By examining the variables PI, RADIUS, and AREA as the program is executing, you can determine that the correct values are being stored. It follows, then, that the error is probably in the expression of the formula for computing the area. To correct the problem, you must edit and recompile the source program, with the exponentiation operator properly specified in the formula expression.

2.1.2 Debugger Command Syntax

Debugger commands resemble other VAX/VMS commands. The general form is:

```
DBG>command [keyword] [operand [,operand] ...]
```

command
Specifies the command name.

keyword
Specifies the qualifiers for SET, SHOW, CANCEL commands.

operand
Specifies the object of the command. The operand may consist of constants, names of variables or array elements, or expressions.

The command, keyword, and operand fields are separated by one or more spaces.

The debugger can process integer, real, double precision, and logical expressions, but not complex or character expressions.

The FORTRAN operators supported by the debugger are listed below, in decreasing order of precedence.

```
*,/
+,-
.NOT.
.AND.
.OR.
.XOR., .EQV.
```

The debugger does not support exponentiation, concatenation, or relational operators.

The debugger evaluates expressions in the same way as FORTRAN. However, the syntax of expressions is slightly different. Spaces are used to separate elements of debugger commands, and are significant to the debugger; therefore, variable names and multicharacter operators (such as .OR.) must contain no embedded spaces.

The debugger accepts constants in the same form used for FORTRAN, with the following exceptions: Hollerith, Radix-50, and octal integer constants (for example "777) are not accepted by the debugger.

Debugger commands observe standard VAX/VMS conventions for abbreviated forms.

Table 2-1 lists the debugger commands and keywords discussed in this chapter, and shows their full and abbreviated forms. Abbreviations are in parentheses next to the full form.

Table 2-1
Debugger Commands and Keywords

Command Names	Keywords
SET (SE)	LANGUAGE (LA)
SHOW (SH)	MODULE (MODU)
CANCEL (CAN)	SCOPE (SC)
EXAMINE (E)	BREAK (B)
EVALUATE (EV)	TRACE (T)
DEPOSIT (D)	WATCH (W)
DEFINE (DEF)	
EXIT (EXI)	
STEP (S)	
GO (G)	
CALL (CA)	

2.1.3 Debugger Symbol Table

The debugger maintains a table of symbols defined by the program with which it is linked. This table provides the name of each symbol defined in the program, its data type, and its address. The table also provides dimension bound information for arrays, and length information for character data.

The debugger's active symbol table provides room for approximately 2000 symbols. Thus, you should pay heed to the number of symbols defined in the programs you are debugging. If your program contains more than one program unit, use the SET MODULE command to be sure the

symbol table contains symbols from the program units you wish to debug. Use the CANCEL MODULE command to remove symbols defined in program units that no longer need debugging. The SET MODULE and CANCEL MODULE commands are defined in Section 2.2.2.

2.2 PREPARING TO DEBUG A PROGRAM

The following sections describe the commands used to establish the proper environment for debugging FORTRAN programs. These commands are:

```
SET LANGUAGE
SHOW LANGUAGE

SET MODULE
SHOW MODULE
CANCEL MODULE

SET SCOPE
SHOW SCOPE
CANCEL SCOPE
```

These commands can be used if the initial settings assumed by the debugger are not appropriate.

2.2.1 SET, SHOW LANGUAGE Command

The SET LANGUAGE command tells the debugger that the debugging dialog is to be conducted according to the conventions of the specified language. For example, if you specify SET LANGUAGE FORTRAN, the debugger will accept and display numeric values in decimal radix.

The command has the form:

```
SET LANGUAGE language
```

language
Specifies the language to be used.

To determine which language is currently in effect, use the SHOW LANGUAGE command. This command has the form:

```
SHOW LANGUAGE
```

The debugger responds by displaying the language in effect. For example:

```
DBG>SHOW LANGUAGE
language: FORTRAN
```

2.2.2 SET, SHOW, CANCEL MODULE Commands

The MODULE commands let you control the contents of the debugger's active symbol table when the program you want to debug consists of multiple program units. These commands perform the following functions:

- SET MODULE places the symbols defined in the specified program unit into the active symbol table. The debugger initializes the active symbol table to include all global symbols, and local symbols of the first program unit specified in the LINK command.
- SHOW MODULE displays the names of all program units whose symbols are potentially available. "Yes" means the symbols for that module are set; "no" means they are not set.
- CANCEL MODULE removes the specified program unit's symbols from the active symbol table.

The SET MODULE command has the form:

```
SET MODULE  {program-unit [,program-unit] ...}
            {/ALL                            }
```

program-unit
Specifies the name of the program unit whose symbols are to be included in the active symbol table.

/ALL
Requests the debugger to set the symbols of all known modules. If there is insufficient space, the debugger displays an error message.

The SHOW MODULE command has the form:

```
SHOW MODULE
```

This command takes no parameters. The debugger responds by displaying the names of the modules linked with the debugger, indicating the modules whose symbols are included in the image, and their sizes. Only those module names marked "Yes" have their symbols in the active symbol table.

The CANCEL MODULE command has the form:

```
CANCEL MODULE  {program-unit [,program-unit] ...}
               {/ALL                            }
```

program-unit
Specifies the name of the program unit for which symbols are to be removed.

/ALL
Specifies that all information is to be purged from the active symbol table.

2.2.3 SET, SHOW, CANCEL SCOPE Commands

The SCOPE commands let you control the default used to resolve references to symbols. When you use a command such as EXAMINE, you can either specify or omit the name of the module in which the symbol

is defined. If you omit the name, the debugger uses a default. If it cannot find the symbol in the default scope, the debugger creates a scope, based on the current value of the PC. This indicates the module or routine in which your program stopped. If that fails, the debugger attempts to find an unambiguous symbol in the remaining program units. A message is displayed if the debugger cannot resolve the reference.

The SCOPE commands perform the following functions:

- SET SCOPE defines the specified program unit to be the default
- SHOW SCOPE displays the current default program unit name
- CANCEL SCOPE revokes the default program unit named previously in a SET SCOPE command

The SET SCOPE command has the form:

```
SET SCOPE program-unit
```

program-unit
Specifies the name of the program unit to be used as the default.

For example:

```
SET SCOPE MAXI
```

The SHOW SCOPE command has the form:

```
SHOW SCOPE
```

This command takes no parameters. The symbol displayed indicates the current scope.

The CANCEL SCOPE command has the form:

```
CANCEL SCOPE
```

This command takes no parameters. Scope becomes <null>.

2.3 CONTROLLING PROGRAM EXECUTION

To see what happens during execution of your program, you must be able to suspend and resume the program at specific points. The following commands are available for these purposes:

```
SET BREAK          SHOW CALLS
SHOW BREAK
CANCEL BREAK       GO
                   STEP
SET TRACE
SHOW TRACE         CTRL/Y
CANCEL TRACE
                   EXIT
SET WATCH
SHOW WATCH
CANCEL WATCH
```

2.3.1 SET, SHOW, CANCEL BREAK Commands

The BREAK commands let you select specified locations for program suspension, so you can examine and/or modify variables or arrays in the program. The BREAK commands perform the following functions.

- SET BREAK defines an address or line number at which to suspend execution
- SHOW BREAK displays all breakpoints currently set in the program
- CANCEL BREAK removes selected breakpoints

The SET BREAK command has the form:

```
SET BREAK[/AFTER:n] address [DO(debugger command(s))]
```

address
Specifies the address at which the breakpoint is to occur. Note that execution is suspended just before the specified address. Section 2.5 describes how addresses are specified.

DO(debugger command(s))
Requests that the debugger perform the specified commands, if any, when the breakpoint is reached.

For example:

```
SET BREAK %LINE 100 DO(EXAMINE TOTAL;  EXAMINE AREA)
```

The result is that the variables TOTAL and AREA are examined when the breakpoint at line 100 is reached.

You can use the /AFTER qualifier to control when a breakpoint takes effect. Thus, if you set a breakpoint on a line that is in the range of a DO loop, and you want the breakpoint to be effective the third time through the loop, then specify the /AFTER switch as shown in the following example:

```
DBG>SET BREAK/AFTER:3 %LINE 20
```

Note that if you use the /AFTER qualifier, the breakpoint is reported the nth time it is encountered, and every time it is encountered thereafter.

The SHOW BREAK command has the form:

```
SHOW BREAK
```

This command takes no parameters. The debugger responds by displaying the location of breakpoints.

The CANCEL BREAK command has the form:

```
CANCEL BREAK {address [,address...]}
             {/ALL                 }
```

address
Removes the breakpoint(s) at the specified address(es).

/ALL
Removes all breakpoints in the program.

2.3.2 SET, SHOW, CANCEL TRACE Commands

The TRACE commands let you set, examine, and remove tracepoints in your program. A tracepoint is similar to a breakpoint in that it suspends program execution, and displays the address at the point of suspension. However, program execution resumes immediately. Thus, tracepoints let you follow the sequence of program execution to ensure that execution is being carried out in the proper order.

Note that tracepoints and breakpoints are mutually exclusive. That is, if you set a tracepoint at the same location as a current breakpoint, the breakpoint will be canceled, and vice versa.

The TRACE commands perform the following functions:

- SET TRACE establishes points within the program at which execution is momentarily suspended
- SHOW TRACE displays the locations in the program at which tracepoints are currently set
- CANCEL TRACE removes one or more tracepoints currently set in the program

The SET TRACE command has the form:

```
SET TRACE address
```

address
Specifies the address at which the tracepoint is to occur.

The SHOW TRACE command has the form:

```
SHOW TRACE
```

This command takes no parameters.

The CANCEL TRACE command has the form:

```
CANCEL TRACE {address [,address ...]}
             {/ALL                  }
```

address
Removes the tracepoint(s) at the specified address(es).

/ALL
Removes all tracepoints in the program.

2.3.3 SET, SHOW, CANCEL WATCH Commands

The WATCH commands let you monitor specified locations to determine when attempts are made to modify their contents, and take the appropriate actions. These locations are called watchpoints. When an attempt is made to change a watchpoint, the debugger halts program execution, displays the address of the instruction, and prompts for a command. Watchpoints are monitored continuously. Thus, you can determine whether locations are being modified inadvertently during program execution.

The WATCH commands perform the following functions:

- SET WATCH defines the location(s) to be monitored
- SHOW WATCH displays the locations currently being monitored
- CANCEL WATCH disables monitoring of specified locations

The SET WATCH command has the form:

```
SET WATCH var
```

var
Specifies the location to be monitored. You can monitor scalar variables and array elements.

For example:

```
SET WATCH AREA
```

Note that watchpoints, tracepoints, and breakpoints are mutually exclusive.

The SHOW WATCH command has the form:

```
SHOW WATCH
```

This command takes no parameters. All watchpoints are displayed.

The CANCEL WATCH command has the form:

```
CANCEL WATCH {var  }
             {/ALL}
```

var
Specifies the location for which monitoring is to be disabled.

/ALL
Removes all watchpoints from the program.

For example:

```
CANCEL WATCH AREA
```

2.3.4 SHOW CALLS Command

This command can be used to produce a traceback of calls, and is particularly useful when you have returned to the debugger following a CTRL/Y command. It has the form:

```
SHOW CALLS [n]
```

The debugger displays a traceback list, showing the sequence of calls leading to the current module. If you include a value for n, the n most recent calls are displayed. The form of the traceback list is described in Section 1.5.

2.3.5 GO, STEP Commands

These commands let you initiate and resume program execution.

- GO initiates execution at a specified location, and continues to conclusion or to the next breakpoint
- STEP initiates execution from the current location, and continues for a specified number of statements

The form of the GO command is:

```
GO [address]
```

address
Specifies the address at which program execution is to begin.

The address parameter is optional; if you omit it, execution starts at the current location.

NOTE

You must not restart a program from the beginning unless you first exit from the debugger. Unspecified results will be produced.

The form of the STEP command is:

```
STEP[/qualifiers] [n]
```

The value specified for n determines the number of statements to be executed. If you specify 0, or omit n, a default of 1 is assumed. Note, however, that if you issue a STEP command while your program is stopped in a module whose symbols are not set in the active symbol table, then n instructions (not statements) will be executed.

You can specify the following qualifiers for the STEP command:

```
/[NO]SYSTEM
/OVER
/INTO
/LINE
/INSTRUCTION
```

/[NO]SYSTEM - If you specify /SYSTEM, you are telling the debugger to count steps wherever they occur, including system address space. The default is /NOSYSTEM.

/OVER - Tells the debugger to ignore calls to subprograms as it steps through the program. That is, it is to step over the call. This is the default.

/INTO - Tells the debugger to recognize calls to subprograms as it steps through the program. That is, it is requested to step into the subprogram.

/LINE - Tells the debugger to step through the program on a line-by-line basis (default for FORTRAN).

/INSTRUCTION - Tells the debugger to step through the program on an instruction-by-instruction basis (default for VAX-11 MACRO).

You can specify these qualifiers each time you issue a STEP command, or you can use a SET STEP command, as shown in the following example:

```
SET STEP INSTRUCTION, INTO, SYSTEM
```

This command specifies that all defaults applicable to FORTRAN programs are to be overridden. When you subsequently issue a STEP command with no qualifiers, these qualifiers are assumed to be in effect. You can, however, supersede them by including a qualifier with a STEP command. Thus,

```
STEP/LINE 10
```

tells the debugger to execute 10 lines, regardless of the SET STEP command.

It is advisable to use STEP to execute only a few instructions at a time. To execute many instructions, and then stop, use a SET BREAK command to set a breakpoint, and then issue a GO command.

2.3.6 CTRL/Y Command

You can use the CTRL/Y command at any time to return to the system command level. This command is issued when you press the CTRL key and the Y key at the same time. The $ prompt will be displayed on the terminal. To return to the debugger, type DEBUG. You can use the CTRL/Y command if your program loops or otherwise fails to stop at a breakpoint. To find out where you were at the instant CTRL/Y was executed, use the SHOW CALLS command after you return to the debugger. See Section 2.3.4.

2.3.7 EXIT Command

The EXIT command lets you exit from the debugger when you are ready to terminate a debugging session. It has the form:

```
EXIT
```

This command takes no parameters. You must use the EXIT command when your program terminates to return to system command level.

2.4 EXAMINING AND MODIFYING LOCATIONS

Once you have set breakpoints and begun program execution, the next step is to see whether correct values are being generated and, possibly, to change the contents of locations as execution proceeds. You may also want to calculate the value of an expression that appears in your program. The debugger provides the following commands for these purposes:

```
EXAMINE

DEPOSIT

EVALUATE
```

2.4.1 EXAMINE Command

The EXAMINE command lets you look at the contents of specified locations. It has the form:

```
EXAMINE [address[:address]]
```

address
Specifies the address whose contents are to be examined; it is usually given symbolically as a variable name or array element name.

Examples:

```
EXAMINE IZZY
```

The contents of variable IZZY are displayed.

```
EXAMINE IARR(I)
```

The contents of the Ith element in array IARR are displayed.

```
EXAMINE IARR(1):IARR(10)
```

The contents of the first through tenth elements of the array IARR are examined.

You can also specify that the contents of an absolute address be displayed. For example:

```
EXAMINE 600
```

The contents of absolute address 600 are displayed.

2.4.2 DEPOSIT Command

The DEPOSIT command lets you change the contents of specified locations. It has the form:

```
DEPOSIT address=value[,value ... ]
```

address
Specifies the address into which the value is to be deposited.

value
Specifies the value to be deposited.

You can change the contents of a specific location, or of several consecutive locations, as shown in the following examples.

```
DEPOSIT IZZY=100
```

This command places the decimal value 100 into the variable IZZY.

```
DEPOSIT IARR(1)=100,150,200
```

This command places the decimal values 100, 150, and 200 into elements 1, 2, and 3 of array IARR.

2.4.3 EVALUATE Command

The EVALUATE command lets you use the debugger as a calculator, to determine the value of expressions. It has the form:

```
EVALUATE expression
```

expression
Specifies the expression whose value is to be determined.

For example:

```
EVALUATE PI*RADIUS
```

The value of this expression will be displayed. You can also use the EVALUATE command to determine addresses, as follows:

```
EVALUATE/ADDRESS expression
```

For example:

```
EVALUATE/ADDRESS I
```

This calculates the address of the variable I, in decimal.

```
EVALUATE/ADDRESS A(J)
```

This calculates the address of the Jth element of array A.

You can also use EVALUATE to perform address arithmetic, such as computing an offset or array element address. For example:

```
EVALUATE/ADDRESS I+4
```

2.5 SPECIFYING ADDRESSES

The debugger allows you to express addresses in symbolic form. Thus, to examine a location, you need only refer to it by its symbolic name. You don't have to concern yourself with its location in memory (unless, of course, you omitted the /DEBUG qualifier from the FORTRAN and LINK commands). Simply specify the variable, array element, or function name in the debugger command.

You also need to tell the debugger where to set breakpoints, watchpoints, and tracepoints. The following sections describe how to specify line numbers, statement labels, and absolute addresses.

2.5.1 Lines, Labels, and Absolute Addresses

Addresses can be specified by line number, statement label, or absolute value. To specify a line number or a statement label, use either a %LINE prefix or a %LABEL prefix, respectively. For example:

```
SET BREAK %LINE 6
```

This command sets a breakpoint at line 6, corresponding to the compiler-generated line numbers shown in the listing. Note that the debugger does not recognize all line numbers, in particular those associated with non-executable statements. If you specify such a line number, the debugger responds with a message indicating that no such

line exists. Simply retry the command, specifying the line number of an executable statement. To specify a statement label, specify a command such as:

```
SET BREAK %LABEL 7
```

This command sets a breakpoint at statement label 7 in the module identified by the current scope (see Section 2.2.3).

To specify an absolute address, do not use a prefix. For example:

```
SET BREAK 700
```

You can also enter absolute addresses in symbolic form. To do so, you must have defined them symbolically, by means of the DEFINE command (see Section 2.5.4).

2.5.2 Specifying Scope

If the program you are debugging consists of more than one program unit, you must be sure that your symbol references are unambiguous. For example, if your main program calls a subroutine, and the symbols from both program units are in the debugger's symbol table, you must distinguish between duplicate symbols.

For example, assume that you want to set a breakpoint in the subroutine, and you issue the following command:

```
SET BREAK %LINE 10
```

Because you do not specify a program unit name in this command, the debugger uses a default to decide which line 10 you mean. If you used a SET SCOPE command, the debugger uses the program unit specified in the SET SCOPE command (see Section 2.2.3). To override this default, you must specify a command in the following general form:

```
SET BREAK %LINE program-unit\10
```

For example:

```
SET BREAK %LINE ARGO\10
```

This command specifically calls for a breakpoint to be set at line 10 in the program unit named ARGO.

Unambiguous references are also required when you specify variables. If there are duplicate variable names (for instance, X) you should specify which X you want, as in the following example:

```
EXAMINE SUB3\X
```

2.5.3 Previous, Current, and Next Locations

The debugger provides a quick method for referring to any of three locations:

- The previous location
- The current location
- The location at the next higher address (next location)

To specify the previous location, type an up-arrow (↑) or circumflex (^). For example:

```
EXAMINE
```

This command displays the contents of the previous location.

To specify the current location, type a dot (.). For example:

```
DEPOSIT .=100
```

This command puts a decimal value of 100 in the current location. This method is most useful after you have looked at a location and decided to change it; or when you want to verify that a DEPOSIT command has been executed as expected.

To specify the next higher location, simply omit the address value entirely. For example:

```
EXAMINE
```

The next location's contents will be displayed.

2.5.4 Defining Addresses Symbolically

You may occasionally need to access absolute addresses. To help you do so, the debugger provides the DEFINE command, which creates a symbolic reference for an absolute address. Then you can refer to the address by its symbolic name, rather than by its absolute value. The DEFINE command has the form:

```
DEFINE name=address
```

For example:

```
DEFINE TOP=1036
```

Subsequent references to this address can be made using the symbol TOP. For example:

```
DEPOSIT TOP=256
```

The contents of address 1036 will be changed to 256.

2.6 CALLING SUBROUTINES FROM THE DEBUGGER

The CALL command lets you call a subroutine from the debugger. It has the form:

```
CALL s[(a, ... )]
```

s
Specifies the subroutine name.

a
Specifies one or more actual arguments.

On return from the subroutine, control returns to the debugger, at the point at which the CALL command was issued. The context (general registers, etc.) that existed at the time of the CALL is also restored.

When calling FORTRAN routines, you must adhere to the FORTRAN calling conventions described in Chapter 5.

2.7 DEBUGGER COMMAND QUALIFIERS

Qualifiers can be used to modify some debugging commands. The general form in which qualifiers are specified is:

```
command/qualifier
```

Qualifiers change the defaults the debugger uses in processing commands. For example, when you deposit a value, the debugger uses decimal radix by default. You can override the default by specifying either /HEX or /OCT. Table 2-2 summarizes the command qualifiers of particular significance in FORTRAN debugging.

Table 2-2
Debugger Command Qualifiers

Qualifier	Function	Commands
/ADDRESS	Indicates that an address value is desired	EVALUATE
/HEX /OCT	Override the default radix (decimal)	EVALUATE DEPOSIT

Refer to the VAX-11 Symbolic Debugger Reference Manual for more information on qualifiers.

2.8 NUMERIC DATA TYPES

The debugger supports all numeric data types used in VAX-11 FORTRAN IV-PLUS, except complex. (Complex values can be deposited and examined, however.) Furthermore, if you attempt to deposit a numeric value into a variable or array element that does not have a matching data type, the value is converted to the data type of the variable or array element.

To deposit a complex value, specify it in two parts as:

```
real part, imaginary part
```

For example:

```
DEPOSIT CPLX=3.4,-4.7
```

When you examine a complex variable or array element, the data is displayed as a complex constant, as (real part, imaginary part).

When you deposit real numbers, you must specify a decimal point. To distinguish single precision and double precision numbers, use E and D, respectively. For example:

Number	Data Type
24.1	Single precision (default)
24.1E0	Single precision
24.1D0	Double precision
241E0	Invalid (no decimal point)

2.9 EFFECTS OF OPTIMIZATION ON DEBUGGING

You should include the /NOOPTIMIZE qualifier when you compile a FORTRAN program that may need to be debugged. This qualifier is necessary because the VAX-11 FORTRAN IV-PLUS compiler performs optimizations by default; and, while highly desirable for bug-free programs, optimization is liable to create difficulty in finding and eliminating bugs from programs in the development stage.

The compiler uses the following optimization techniques:

- Using central processor condition codes
- Binding frequently-used variables to registers
- Assuming that the flow of control proceeds in a certain sequence, based on source code

These techniques and some of the implications for debugging are described below.

2.9.1 Use of Condition Codes

This optimization technique takes advantage of the way in which the central processor's condition codes are set. For example, consider the following source code:

```
X = X + 2.5
IF (X .LT.  0) GO TO 20
```

Rather than test the new value of X to determine whether to branch, the optimized object code bases its decision on the condition code settings after 2.5 is added to X. Thus, if you attempt to set a breakpoint at the second line, and deposit a different value into X, you will not achieve the intended result, because the condition codes no longer reflect the value of X. In other words, the decision to branch is being made without regard to the new value of the variable.

2.9.2 Register Binding

This technique is used to reduce the number of memory references or load-and-store instructions needed. The values of frequently-used variables are kept in general registers, and the registers are used, rather than the variables. Therefore, if you deposit a new value in a

variable that has been bound to a register, the new value will have no effect. Moreover, if you examine the variable, the current value (which is kept in the register) may not be displayed.

2.9.3 Control Flow

The compiler assumes that statements will be executed in the sequence in which they appear in source code, if there are no intervening labels. Optimization of such code sequences will not let you use the "GO address" version of the GO command.

2.9.4 Effects of /NOOPTIMIZE and /OPTIMIZE

The /NOOPTIMIZE qualifier tells the compiler not to assume that condition codes are valid; not to keep the values of variables in general registers; and not to optimize across statement boundaries. In short, the object program directly reflects the source program. When /NOOPTIMIZE is in effect, you can issue any of the debugging commands.

When /OPTIMIZE is in effect, you should not use the GO address command. However, you can set and clear breakpoints and examine COMMON variables. If you need to debug a program that was compiled with /OPTIMIZE in effect, you may need a compiler listing of the generated machine code. Thus, if you do not suppress optimization, you should specify /LIST and /MACHINE_CODE in the FORTRAN command.

CHAPTER 3

FORTRAN INPUT/OUTPUT

This chapter describes FORTRAN input/output (I/O) as implemented for VAX-11 FORTRAN IV-PLUS. In particular, it provides information about FORTRAN IV-PLUS I/O in relation to VAX-11 Record Management Services (RMS). The topics covered include:

- VAX/VMS file specifications (Section 3.1)
- Logical names as used in FORTRAN (Section 3.2)
- FORTRAN file characteristics (Section 3.3)
- FORTRAN record formats (Section 3.4)
- OPEN statement features (Section 3.5)
- Auxiliary I/O operations (Section 3.6)
- Local interprocess communication by means of mailboxes (Section 3.7)
- Remote communication by means of DECnet-VAX (Section 3.8)

The FORTRAN I/O statements are: READ, WRITE, ACCEPT, PRINT, and TYPE. The device or file to or from which data is transferred is designated by a logical unit number, specified or implied as part of the I/O statement. Logical unit numbers are integers from 0 to 99.

For example:

```
READ (2,100) I,X,Y
```

This statement specifies that data is to be entered from the device or file corresponding to logical unit 2, in the format specified by the FORMAT statement labeled 100.

The association between the logical unit number and the physical device or file occurs at execution time. You can change this association at execution time, if necessary, to match the needs of the program and the available resources. You need not change the logical unit numbers specified in the program. Thus, FORTRAN programs are inherently device independent.

You can use standard FORTRAN I/O statements to communicate between processes on either the same computer or different computers. Mailboxes permit interprocess communication on the same computer. DECnet network facilities are used for interprocess communication on different computers. DECnet can also be used to process files on different computers.

3.1 FILE SPECIFICATION

A complete VAX/VMS file specification has the form:

```
node::device:[directory]filename.filetype.version
```

For example:

```
BOSTON::DBA0:[SMITH]TEST.DAT.2
```

node
BOSTON

device
DBA0 (unit 0 of disk DBA)

directory
SMITH (the file name is cataloged in the disk directory named SMITH)

filename
TEST

filetype
DAT

version number
2

If you omit elements of the file specification, the system supplies default values, as follows. If you omit the node, the local computer is used; if you omit the device or directory, the current user default is used; if you omit the file name, the system supplies FOR0nn, where nn is the logical unit number; if you omit the file type, the system supplies DAT; and if you omit the version number, the system supplies either the highest current version number (for input) or the highest current version number plus 1 (for output).

For example, suppose your default device is DBA0: and your default directory is SMITH, and you specify:

```
READ (8,100)
```

The default file specification is:

```
DBA0:[SMITH]FOR008.DAT.n
```

The value of n equals the highest current version number of FOR008.DAT.

Then, suppose you specify:

```
WRITE (9,200)
```

The default file specification is:

```
DBA0:[SMITH]FOR009.DAT.m
```

Where m is 1 greater than the highest existing version number of FOR009.DAT.

3.2 LOGICAL NAMES

The VAX/VMS operating system provides the logical name mechanism as a means of associating logical units with file specifications. A logical name is an alphanumeric string, up to 15 characters long, that is specified instead of a file specification.

The operating system provides a number of predefined logical names, already associated with particular file specifications. Table 3-1 lists the logical names of special interest to FORTRAN users.

Table 3-1
Predefined System Logical Names

Name	Meaning	Default
SYS$DISK	Default device and directory	As specified by the user
SYS$INPUT	Default input stream	User's terminal (interactive); batch command file (batch)
SYS$OUTPUT	Default output stream	User's terminal (interactive); batch log file (batch)

You can create a logical name dynamically, and associate it with a file specification by means of the VAX/VMS ASSIGN command. Thus, before program execution, you can associate the logical names in your program with the file specification appropriate to your needs.

For example:

```
$ ASSIGN  DBA0:[SMITH]TEST.DAT.2  LOGNAM
```

This command creates the logical name LOGNAM and associates it with the file specification DBA0:[SMITH]TEST.DAT.2. This will be the file specification used when the logical name LOGNAM is encountered during program execution.

Logical names provide great flexibility because they can be associated not only with a complete file specification, but with a device, a device and a directory, or even another logical name.

3.2.1 FORTRAN Logical Names

Usually, FORTRAN I/O is performed by associating a logical unit number with a device or file. The VAX/VMS logical name concept allows one more level of association: a user-specified logical name can be associated with a logical unit number.

VAX-11 FORTRAN IV-PLUS provides predefined logical names, in the form:

```
FOR0nn
```

The value of nn corresponds to the logical unit number. By default, each FORTRAN logical name is associated with a file named FOR0nn.DAT,

which is assumed to be located on your default disk, under your default directory. For example:

```
WRITE (17,200)
```

If you enter this statement, without including an explicit file specification, the data will be written to your default disk, to a file named FOR017.DAT, under your default directory.

You can change the file specification associated with a FORTRAN logical unit number by using the ASSIGN command to change the file associated with the corresponding FORTRAN logical name. For example:

```
$ ASSIGN  DBA0:[SMITH]TEST.DAT.2  FOR017
```

This command associates the FORTRAN logical name FOR017 (and therefore logical unit 17) with file TEST.DAT.2 on device DBA0, in directory SMITH.

You can also associate the FORTRAN logical names with any of the predefined system logical names. Two examples follow.

1. `$ ASSIGN  SYS$INPUT  FOR005`

   This command associates logical unit 5 with the default input device (for example, the batch input stream).

2. `$ ASSIGN  SYS$OUTPUT  FOR006`

   This command associates logical unit 6 with the default output device (for example, the batch output stream).

Many VAX-11 systems provide system-wide default logical name assignments for logical units 5 and 6 as shown in the preceding example.

3.2.2 Implied FORTRAN Logical Unit Numbers

The READ, ACCEPT, PRINT, and TYPE statements do not include an explicit logical unit number. Each of these FORTRAN statements uses an implicit logical unit number and logical name. Each of these logical names is, in turn, associated with one of the system's predefined logical names, by default. Table 3-2 shows these relationships.

Table 3-2
Implicit FORTRAN Logical Units

Statement	FORTRAN Logical Name	System Logical Name
READ f,list	FOR$READ	SYS$INPUT
ACCEPT f,list	FOR$ACCEPT	SYS$INPUT
PRINT f,list	FOR$PRINT	SYS$OUTPUT
TYPE f,list	FOR$TYPE	SYS$OUTPUT

As with any other FORTRAN logical name, you can change the file specifications associated with these FORTRAN logical names by means of the ASSIGN command. For example:

```
$ASSIGN  DBA0:[SMITH]TEST.DAT.2  FOR$READ
```

Following execution of this command, the READ statement's logical name (FOR$READ) will refer to the file TEST.DAT.2, on device DBA0, in directory SMITH.

3.2.3 OPEN Statement NAME Keyword

You can use the NAME keyword of the OPEN statement to specify the particular file to be opened on a logical unit. (Section 3.5 describes the OPEN statement in greater detail.) For example:

```
OPEN (UNIT=4, NAME='DBA0:[SMITH]TEST.DAT.2', TYPE='OLD')
```

In this example, the file TEST.DAT.2, on device DBA0:, in directory SMITH, is opened on logical unit 4. Neither the default file specification (FOR004.DAT) nor the FORTRAN logical name FOR004 is used. The value of the NAME keyword can be a character constant, variable, or expression.

You can also specify a logical name as the value of the NAME keyword, if the logical name is associated with a file specification. For example:

```
$ASSIGN  DBA0:[SMITH]TEST.DAT  LOGNAM
```

This command assigns the logical name LOGNAM to the file specification DBA0:[SMITH]TEST.DAT. The logical name can then be used in an OPEN statement, as follows:

```
OPEN (UNIT=19,NAME='LOGNAM',TYPE='OLD')
```

When an I/O statement refers to logical unit 19, the system uses the file specification associated with logical name LOGNAM.

If the value specified for the NAME keyword has no associated file specification, it is regarded as a true file name rather than as a logical name. That is, if LOGNAM had not been previously associated with the file specification DBA0:[SMITH]TEST.DAT by means of an ASSIGN command, then the following statement would indicate that a file named LOGNAM.DAT is located on the default device, in the default directory:

```
OPEN (UNIT=19,NAME='LOGNAM',TYPE='OLD')
```

A logical name specified in an OPEN statement must not contain brackets or periods. The system treats any name containing these punctuation marks as a file specification, not as a logical name.

3.2.4 Assigning Files to Logical Units

You can assign files to logical units in any of three ways:

1. By using default logical names; two examples follow.

   ```
   READ (7,100)
   ```

   Logical unit FOR007 is associated with the file FOR007.DAT by default.

   ```
   TYPE 100
   ```

   Logical unit FOR$TYPE is associated with SYS$OUTPUT by default.

2. By specifying a logical name in an OPEN statement. For example:

   ```
   OPEN (UNIT=7,NAME='LOGNAM')
   ```

3. By supplying a file specification in an OPEN statement. For example:

   ```
   OPEN (UNIT=7,NAME='LOGNAM.DAT')
   ```

You use the ASSIGN command to change the association of logical names and file specifications.

A logical name used with the NAME keyword of the OPEN statement must be associated with a file specification, and the character expression specified for the NAME keyword must contain no punctuation marks. Otherwise, the logical name will be treated as a true file specification.

Use the VAX/VMS SHOW LOGICAL command to determine the current associations of logical names and file specifications.

To remove the association of a logical name and a file specification, use the DEASSIGN command, in the form:

```
$DEASSIGN logical-name
```

3.2.5 Assigning Logical Names with MOUNT Commands

You can specify a logical name as a parameter of the MOUNT command. The MOUNT command has the form:

```
$ MOUNT device-name,... [volume-label,...] [logical-name[:]]
```

If your program refers to devices by means of logical names, you can change the association between the device name and the logical name when you mount the device. For example:

```
$ MOUNT MT: TAPE2 MYTAPE
```

This command associates the logical name MYTAPE with device name MT and volume label TAPE2. Whenever your program refers to logical name MYTAPE, access will be to the volume labeled TAPE2 mounted on the default magnetic tape unit. If you subsequently mount a different tape to be referenced by the logical name MYTAPE, you can change the logical name association when you issue the MOUNT command. For example:

```
$ MOUNT MT: TAPE7 MYTAPE
```

3.3 FILE CHARACTERISTICS

A clear distinction must be made between the way in which files are organized and the manner in which records are accessed.

The term "file organization" applies to the way records are physically arranged on a storage device. "Record access" refers to the method used to read records from or write records to a file, regardless of its organization. A file's organization is specified when the file is created, and cannot be changed. Record access is specified each time the file is opened, and can be different each time.

3.3.1 File Organization

VAX-11 FORTRAN IV-PLUS supports two file organizations:

- Sequential
- Relative

The organization of a file is specified by means of a keyword in the OPEN statement, as described in Section 3.5.4.

3.3.1.1 **Sequential Organization** - The default file organization is sequential.

Sequential files consist of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, etc.). As a result, records can be added only at the end of the file. Sequential file organization is permitted on all devices supported by the VAX-11 FORTRAN system.

3.3.1.2 **Relative Organization** - Relative files are permitted only on disk devices. A relative file consists of numbered positions, called cells. These cells are of fixed, equal length, and are numbered consecutively from 1 to n, where 1 is the first cell, and n is the last available cell in the file.

This arrangement lets you place records into the file according to cell number; the cell number becomes the record's relative record number; that is, its location relative to the beginning of the file. As a result, you can retrieve records directly by specifying their relative record number, because the actual location of the record is easily calculated relative to the beginning of the file. You can add records to, or delete them from, the file regardless of their

location, as long as you keep track of the relative record numbers of the records.

3.3.2 Access to Records

Records can be accessed in two ways:

- Sequential access
- Direct access

The access mode chosen is unrelated to the file organization. You can access records in both relative and sequential files sequentially or directly (with certain restrictions, described below).

3.3.2.1 **Sequential Access** - If you select sequential access mode, records are written to or read from the file, starting at the beginning and continuing through the file one record after another.

Sequential access to a file means that a particular record can be retrieved only when all the records preceding it have been read. Writing records by means of sequential access varies according to the file organization. New records can be written only at the end of a sequentially organized file. For a relative organization file, however, a new record can be written at any point, replacing the existing record in that cell. For example, if two records are read, and then a record is written, the new record occupies cell 3 of the file.

3.3.2.2 **Direct Access** - If you select direct access mode, you determine the order in which records are read or written. Each READ or WRITE statement must include the relative record number indicating the record to be read or written.

You can access relative files directly, and you can also directly access a sequential file if it contains fixed length records and resides on disk. Because direct access uses cell numbers to find records, you can issue successive READ or WRITE statements requesting records that either precede or follow previously requested records.

For example:

```
READ (12'24) - read record 24 in file 12

READ (12'20) - read record 20 in file 12
```

3.4 RECORD STRUCTURE

Records are stored in one of three formats:

1. Fixed length
2. Variable length
3. Segmented

Fixed length and variable length formats can be used with sequential or relative file organization. Segmented format is unique to FORTRAN, and can be used only with sequential file organization, and only for unformatted sequential access. You should not use segmented records for files that will be read by programs written in languages other than FORTRAN.

3.4.1 Fixed Length Records

When you specify fixed length records (see Section 3.5.7), you are specifying that all records in the file contain the same number of bytes. When you create a file that is to contain fixed length records, you must specify the record size (see Section 3.5.6). A sequentially organized file opened for direct access must contain fixed length records, to allow the record number to be computed correctly. Note that in a relative organization file each fixed length record contains an extra byte, the deleted-record control byte.

3.4.2 Variable Length Records

Variable length records can contain any number of bytes, up to a specified maximum. Variable length records are prefixed by a count field, indicating the number of bytes in the record. The count field comprises two bytes on a disk device, and four bytes on magnetic tape. The value stored in the count field indicates the number of data bytes in the record. Variable length records in relative files are actually stored in fixed length cells, the size of which must be specified by means of the RECORDSIZE keyword of the OPEN statement (see Section 3.5.6). This value specifies the largest record that can be stored in the file. Each variable length record in a relative file contains three extra bytes, two for the count field and one for deleted record control.

The count field of a variable length record is available when you read the record; issue a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

3.4.3 Segmented Records

A segmented record is a single logical record consisting of one or more variable length records. Each variable length record constitutes a segment. The length of a segmented record is arbitrary. Segmented records are useful when you want to write exceptionally long records, and are especially appropriate to sequentially organized files. Unformatted sequential records written to sequentially organized files are, by default, stored as segmented records.

Because there is no set limit on the size of a segmented record, each variable length record in the segmented record contains control information to indicate that it is one of the following:

- The first segment in the segmented record
- The last segment in the segmented record
- The only segment in the segmented record
- None of the above

This control information is contained in the first two bytes of each segment of a segmented record. Thus, when you wish to access an unformatted sequential file that contains fixed length or variable length records, you must specify RECORDTYPE='FIXED' or 'VARIABLE'(as appropriate) when you open the file. Otherwise the first two bytes of each record will be misinterpreted as control information, and errors will probably result.

3.5 OPEN STATEMENT KEYWORDS

The following sections supplement the OPEN statement description that appears in the VAX-11 FORTRAN IV-PLUS Language Reference Manual. In particular, implementation-dependent and/or system-dependent aspects of certain OPEN statement keywords are described as affected by the VAX-11 Record Management Services (RMS) implementation. For more information refer to the VAX-11 Record Management Services Reference Manual.

3.5.1 BLOCKSIZE Keyword

The BLOCKSIZE keyword specifies the physical I/O transfer size for the file. It has the form:

```
BLOCKSIZE = bks
```

For magnetic tape files, the value of bks specifies the physical record size in the range 18 to 32767 bytes. The default value is 2048 bytes.

For sequential disk files, the value of bks is rounded up to an integral number of 512-byte blocks and used to specify RMS multiblock transfers. The number of blocks transferred can be 1 to 127. The default value is 2048 bytes.

For relative files, the value of bks is rounded up to an integral number of 512-byte blocks and used to specify the RMS bucket size, in the range 1 to 32 blocks. The default is the smallest value capable of holding a single record.

3.5.2 BUFFERCOUNT Keyword

The BUFFERCOUNT keyword specifies the number of memory buffers to use. It has the form:

```
BUFFERCOUNT = bc
```

The range of values for bc is from 1 to 255. The size of each buffer is determined by the BLOCKSIZE keyword. Thus, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144. The default is two buffers for sequential files and one buffer for relative files.

3.5.3 INITIALSIZE and EXTENDSIZE Keywords

The INITIALSIZE keyword specifies the initial storage allocation for a disk file, and the EXTENDSIZE keyword specifies the amount by which a disk file is extended each time more space is needed for the file.

INITIALSIZE is effective only at the time the file is created. If EXTENDSIZE is specified when the file is created, the value specified is the default value used to allocate additional storage for the file. If you specify EXTENDSIZE when you open an existing file, the value you specify supersedes any EXTENDSIZE value specified when the file was created, and remains in effect until you close the file. Unless specifically overridden, the default EXTENDSIZE value is in effect on subsequent openings of the file.

The system attempts to allocate contiguous space for INITIALSIZE. If not enough contiguous space is available, noncontiguous space is allocated.

3.5.4 ORGANIZATION Keyword

The ORGANIZATION keyword specifies file organization. It has the form:

```
ORGANIZATION = {'RELATIVE'  }
               {'SEQUENTIAL'}
```

The default file organization is sequential.

When an existing file is opened, the actual organization of the file is used.

The relative file organization is applicable mainly when creating files to be used in non-FORTRAN applications, or when reading relative files created by programs written in languages other than FORTRAN.

3.5.5 READONLY Keyword

The READONLY keyword specifies that write operations are not allowed on the file being opened. The FORTRAN I/O system's default file access privileges are read-write, which can cause run-time I/O errors if the file protection does not permit write access. The READONLY keyword has no effect on the protection specified for a file. Its main purpose is to allow a file to be read simultaneously by two or more programs. Thus, if you wish to open a file for the purpose of reading the file, but do not want to prevent others from being able to read the same file while you have it open, specify the READONLY keyword.

3.5.6 RECORDSIZE Keyword

The RECORDSIZE keyword specifies how much data can be contained in a record. It has the form:

```
RECORDSIZE = rl
```

The value specified for rl indicates the length of the logical records in the file. For files that contain fixed length records, rl specifies the size of each record; for files that contain variable length records, rl specifies the maximum length for any record.

The value of rl does not include the two segment control bytes (if present), or the bytes that RMS requires for maintaining record length and deleted-record control information (two or four for sequential organization, and one or three for relative organization).

The value of rl is interpreted as either bytes or longwords, depending on whether the records are formatted (bytes) or unformatted (longwords, that is, 4-byte units). Table 3-3 summarizes the maximum values that can be specified for rl, based on file organization and record format.

Table 3-3
RECORDSIZE Limits

File Organization	Record Format	
	Formatted (bytes)	Unformatted (longwords)
Sequential	32766	8191
Sequential and variable length records on ANSI magnetic tape	9999*	2499*
Relative	16380	4095

* Limit imposed by 4-byte ASCII count field.

If you are opening an existing file containing fixed length records or that has relative organization, and you specify a value for RECORDSIZE that is different from the actual length of the records in the file, an error occurs. If you omit RECORDSIZE when opening an existing file, the record length specified when the file was created is used by default.

You must specify RECORDSIZE when you create a file that is to contain fixed length records or that has relative organization.

3.5.7 RECORDTYPE Keyword

The RECORDTYPE keyword specifies the structure of records in a file. It has the form:

$$\text{RECORDTYPE} = \left\{ \begin{matrix} \text{'FIXED'} \\ \text{'VARIABLE'} \\ \text{'SEGMENTED'} \end{matrix} \right\}$$

This keyword is particularly useful when you want to override the default record structure used to create a file. The default record structure is:

FIXED - direct access, sequential, and relative

VARIABLE - formatted sequential

SEGMENTED - unformatted sequential

The default used when accessing an existing file is the record structure of the file, except for unformatted sequential files containing fixed or variable length records. In this case, you must explicitly override the default (SEGMENTED) by specifying the appropriate RECORDTYPE value in the OPEN statement. You cannot use an unformatted READ statement to access an unformatted sequentially organized file that contains fixed length or variable length records, unless you specify the corresponding RECORDTYPE value in your OPEN statement. Files containing segmented records can be accessed only by unformatted sequential FORTRAN I/O statements.

3.5.8 SHARED Keyword

The SHARED keyword specifies that the file can be accessed by more than one program at a time, or by the same program on more than one logical unit. The forms of sharing permitted depend on the organization of the file.

For sequential files, both read and write sharing are permitted. Because RMS does not prevent two or more programs from accessing the same file simultaneously, however, user programs that share write access to a file must provide interprocess communication and coordination to ensure reliable performance. Otherwise, problems may develop. For example, if two programs write to a shared file that contains records that cross block boundaries, records containing data written by two different programs can result. This can happen if the co-operating programs do not coordinate their read, modify, and rewrite sequences, which are otherwise asynchronously and independently performed.

Furthermore, RMS usually tries to minimize disk activity by postponing a rewrite in case a subsequent read or write can be performed using the program's buffer image. Thus, the file's disk image may be out of date for arbitrary time intervals. This problem can occur for both sequential and direct access I/O.

You can encounter a similar problem involving the logical end-of-file on disk. When a file is extended, the logical end-of-file in the disk image is not updated until the file is closed. This means that if a file is open and program A is adding new records to it, and program B opens the same file before program A has closed the file, program B cannot read the new records even after program A finishes and closes the file. Program B can read the new records only by closing and reopening the file. Only then will the file's disk image reflect the updated end-of-file.

Relative files permit no write sharing. For shared reading to occur, all programs that open the file must specify the READONLY keyword.

3.5.9 USEROPEN Keyword

The USEROPEN keyword provides access to RMS features not directly supported by the FORTRAN I/O system. That is, this keyword allows access to RMS capabilities, while retaining the ease and convenience of FORTRAN programming. The USEROPEN keyword is intended for experienced users.

For the interface specification for a USEROPEN routine, see the VAX-11 Common Run-Time Procedure Library Reference Manual.

3.6 AUXILIARY I/O OPERATIONS

This section describes implications of the following I/O statements:

```
FIND
BACKSPACE
ENDFILE
```

A FIND statement is similar to a direct access READ statement with no I/O list, and can result in an existing file being opened. An associated variable will be set to the specified record number.

A BACKSPACE statement cannot be performed on a file that is opened for append access, because of the manner in which backspacing is done. A backspace operation requires that the current record count be available to the FORTRAN I/O system, because backspacing from record n is done by rewinding to the start of the file and then performing n-1 successive reads to reach the previous record. If the file is open for append access, the current record count is not available to the FORTRAN I/O system.

The ENDFILE statement writes an end-file record. The following convention has been adopted, since RMS does not support the embedded end-file concept: an end-file record is a 1-byte record that contains the hexadecimal code 1A (CTRL/Z). An end-file record can be written only to sequentially organized files that are accessed as formatted sequential or unformatted segmented sequential. End-file records should not be written in files that will be read by programs written in a language other than FORTRAN.

3.7 LOCAL INTERPROCESS COMMUNICATION: MAILBOXES

It is often useful to exchange data between processes; for example, to synchronize execution, or to send messages.

A mailbox is a record-oriented pseudo I/O device that allows data to be passed from one process to another. Mailboxes are created by the Create Mailbox system service. The following sections describe how to create mailboxes and how to send and receive data using mailboxes.

3.7.1 Creating a Mailbox

Use the Create Mailbox system service to create a mailbox, as follows:

```
INTEGER*2 ICHAN
INTEGER*4 SYS$CREMBX
MAILBX= SYS$CREMBX(,ICHAN, , , , ,'MAILBOX')
```

The INTEGER*2 variable ICHAN is used to store the number of the mailbox, which is returned by the Create Mailbox and Assign Channel system services. This argument is required by the Create Mailbox systems service, so you must specify an INTEGER*2 variable such as ICHAN. However, all subsequent references to the mailbox are by logical name.

For more information about calling system services, see Chapter 5. For more information about the arguments supplied to the Create Mailbox system service, see the VAX/VMS System Services Reference Manual.

3.7.2 Sending and Receiving Data Using Mailboxes

Sending or receiving data to or from a mailbox is no different from other forms of FORTRAN I/O. The mailbox is simply treated as a record-oriented I/O device.

Use FORTRAN formatted sequential I/O statements to send and receive messages. Use WRITE statements to send data and READ statements to receive data.

Data transmission by means of mailboxes is performed synchronously, so that communicating processes can be synchronized. That is, a program that writes a message to a mailbox waits until the message is read, and a program that reads messages from a mailbox waits until a message is written. When the writing program closes the mailbox, an end-of-file condition is returned to the reading program.

The sample program below reads messages from a mailbox known by the logical name MAILBOX. The messages comprise file names, which the program reads. The program then prints the file associated with the file names.

```
            CHARACTER FILNAM*64,TEXT*133
            OPEN(UNIT = 1, NAME = 'MAILBOX', TYPE = 'OLD')
1           READ (1,100,END=12)FILNAM
100         FORMAT (A)

            OPEN(UNIT = 2, NAME = FILNAM, TYPE = 'OLD')
            OPEN(UNIT = 3, NAME = 'SYS$OUTPUT')
2           READ(2,100, END = 10) TEXT
            WRITE(3,100) TEXT
            GO TO 2

10          CLOSE(UNIT = 2)
            CLOSE(UNIT = 3)
            GO TO 1
12          END
```

3.8 COMMUNICATING WITH REMOTE COMPUTERS: NETWORKS

If your system supports DECnet-VAX facilities, and your computer is one of the nodes in a DECnet-VAX network, you can communicate with other nodes in the network by means of standard FORTRAN I/O statements. These statements let you exchange data with a program at the remote computer (task-to-task communication), and to access files at the remote computer (resource sharing).

Both task-to-task communication and file access between systems are transparent. That is, there is no apparent difference between these intersystem exchanges, and local interprocess and file access exchanges.

To invoke network communication, specify a node name as the first element of a file specification. For example:

```
BOSTON::DBA0:[SMITH]TEST.DAT.2
```

For remote task-to-task communication, you must use a special form of file specification: you must use TASK_ in place of the device name, and use the task name in place of the file name. For example:

```
BOSTON::TASK_:UPDATE
```

The following example shows how messages can be sent to and received from a remote program by means of standard FORTRAN I/O statements.

```
          OPEN (UNIT=7,NAME='BOSTON::TASK_:UNA',ERR=200)
          READ (7,100)IARRAY
100       FORMAT (20I8)
          CALL STATS(IARRAY)
          WRITE (7,100)IARRAY
200       CLOSE (UNIT=7)
          END
```

The effect of these statements is to establish a link with a job (task) named UNA at the node BOSTON, and receive data from the logical unit (7) associated with the remote program. The data is stored in a 20-element array, and a call is issued to the subroutine STATS, which processes the data. The results are then sent back to BOSTON, and the link is broken.

The following example shows how a remote file can be updated by means of standard FORTRAN I/O statements.

```
          CHARACTER*64 DATA
          OPEN(UNIT = 2, NAME = 'DENVER::MASTER.DAT',
          1 ACCESS = 'DIRECT', TYPE = 'OLD')

     1    READ(1,100,END = 2) IREC, DATA
     100  FORMAT(I10, A)

          WRITE(2'IREC) DATA
          GOTO 1

     2    CLOSE(UNIT = 1)
          CLOSE(UNIT = 2)
          END
```

This program reads local data describing transactions, and writes the new records into the remote file.

If you use logical names in your program, you can equate the logical names with either local or remote files. Thus, if your program normally accesses a remote file, and the remote node becomes unavailable, you can bring the volume set containing the file to the local site. You can then mount the volume set, and assign the appropriate logical name. For example:

Remote Access

```
$ ASSIGN REM::APPLIC_SET:file-name  LOGIC
```

Local Access

```
$ MOUNT device-name APPLIC_SET
$ ASSIGN APPLIC_SET:file-name  LOGIC
```

The MOUNT and ASSIGN commands are described in detail in the VAX/VMS Command Language User's Guide.

DECnet facilities are described fully in the DECnet-VAX Reference Manual.

CHAPTER 4

USING CHARACTER DATA

The FORTRAN character data type allows you to easily manipulate alphanumeric data. You can use character data in the form of character variables, arrays, constants, and expressions. A character operator (//) is available to form character strings by concatenating the character elements in a character expression. See Section 4.2.

4.1 CHARACTER SUBSTRINGS

You can select certain segments (substrings) from a character variable or array element by specifying the variable name, followed by delimiter values indicating the leftmost and/or rightmost characters in the substring. For example, if the character string NAME contained:

```
ROBERTΔWILLIAMΔBOBΔJACKSON
```

and you wished to extract the substring BOB, you would specify the following:

```
NAME(16:18)
```

If you omit the first value, you are indicating that the first character of the substring is the first character in the variable. For example, if you specify

```
NAME(:18)
```

the resulting substring is

```
ROBERTΔWILLIAMΔBOB
```

If you omit the second value, you are specifying the rightmost character to be the last character in the variable. For example:

```
NAME(16:)
encompasses BOBΔJACKSON
```

4.2 BUILDING CHARACTER STRINGS

It is sometimes useful to create strings from two or more separate strings. This is done by means of the concatenation operator; the double slash (//). For example, you might wish to create a variable called NAME, consisting of the following strings:

```
FIRSTNAME
MIDDLENAME
NICKNAME
LASTNAME
```

To do so, define each as a character variable of a specified length. For example:

```
CHARACTER*42 NAME
CHARACTER*12 FIRSTNAME,MIDDLENAME,LASTNAME
CHARACTER*6 NICKNAME
```

Concatenation is accomplished as follows:

```
NAME = FIRSTNAME//MIDDLENAME//NICKNAME//LASTNAME
```

Thus, if the strings contained the values:

```
FIRSTNAME = 'ROBERT'
MIDDLENAME = 'WILLIAM'
NICKNAME = 'BOB'
LASTNAME = 'JACKSON'
```

which are stored individually as

```
ROBERTΔΔΔΔΔΔ
WILLIAMΔΔΔΔΔ
BOBΔΔΔ
JACKSONΔΔΔΔΔ
```

then, when concatenated and stored in NAME, they become the string:

```
ROBERTΔΔΔΔΔΔWILLIAMΔΔΔΔΔBOBΔΔΔJACKSONΔΔΔΔΔ
```

Applying the substring extraction facility, you can get the stored nickname by specifying:

```
NAME(25:30)
```

which picks up the 6-character NICKNAME substring (including trailing blanks). Thus BOBΔΔΔ is retrieved as the substring.

4.3 CHARACTER CONSTANTS

Strings of alphanumeric characters enclosed in apostrophes are character constants. You can assign a character value to a character variable in much the same way as you would assign a numeric value to a real or integer variable. For example:

```
XYZ = 'ABC'
```

As a result of this statement, the characters ABC are stored in location XYZ. Note that if XYZ's length is less than three bytes, the character string will be truncated on the right. Thus if you specified

```
CHARACTER*2 XYZ

XYZ = 'ABC'
```

The result is AB. If, on the other hand, the variable is longer than the constant, it is padded on the right with blanks. For example:

```
CHARACTER*6 XYZ

XYZ = 'ABC'
```

results in

```
ABCΔΔΔ
```

being stored in XYZ. If the previous contents of XYZ were CBSNBC, the result would still be ABCΔΔΔ: the previous contents are overwritten.

You can give character constants symbolic names by using the PARAMETER statement. For example:

```
PARAMETER TITLE = 'THE METAMORPHOSIS'
```

The symbolic name TITLE can then be used anywhere a character constant is allowed.

Note that an apostrophe can be included as part of the constant. To do so, specify two consecutive apostrophes. For example:

```
PARAMETER TITLE = 'FINNEGAN''S WAKE'
```

results in the character constant FINNEGAN'S WAKE.

4.4 DECLARING CHARACTER DATA

To declare variables or arrays as character type, use the CHARACTER type declaration statement, as shown in the following example:

```
CHARACTER*10 TEAM(12),PLAYER
```

This statement defines a 12-element character array (TEAM), each element of which is 10 bytes long; and a character variable (PLAYER), which is also 10 bytes long.

You can specify different lengths for variables in a CHARACTER statement by including a length value for specific variables. For example:

```
CHARACTER*6 NAME,AGE*2,DEPT
```

In this example, NAME and DEPT are defined to be 6-byte variables, while AGE is defined to be a 2-byte variable.

4.5 INITIALIZING CHARACTER VARIABLES

Use the DATA statement to preset the value of a character variable. For example:

```
CHARACTER*10 NAME,TEAM(5)
DATA       NAME/' '/,TEAM/'SMITH','JONES',
               'DOE','BROWN','GREEN'/
```

Note that NAME will contain 10 blanks, while each array element in TEAM will contain the corresponding character value, right-padded with blanks.

To initialize an array so that each of its elements contains the same value, use a DATA statement of the following type:

```
CHARACTER*5 TEAM(10)
DATA TEAM/10*'WHITE'/
```

The result is a 10-element array in which each element contains WHITE.

4.6 PASSED LENGTH CHARACTER ARGUMENTS

Subprograms that manipulate character data can be written to accept character actual arguments of any length by specifying the length of the dummy argument as passed length. To indicate a passed length dummy argument, use an asterisk (*) as follows:

```
SUBROUTINE REVERSE(S)
CHARACTER*(*) S
  .
  .
  .
```

The passed length notation indicates that the length of the actual argument is used when processing the dummy argument string. This length can change from one invocation of the subprogram to the next. For example:

```
CHARACTER  A*20,B*53
  .
  .
  .
CALL REVERSE(A)
CALL REVERSE(B)
```

In the first call to REVERSE, the length of S will be 20; in the second call, its length will be 53.

The FORTRAN function LEN can be used to determine the actual length of the string (see Section 4.8.4).

4.7 CHARACTER DATA EXAMPLES

An example of character data usage is shown below. The example (Figure 4-1) is a program that manipulates the letters of the alphabet. The results are shown in Figure 4-2.

```
C
C       CHAREXMPL.FOR
C
C       CHARACTER DATA TYPE EXAMPLE PROGRAM FOR VAX FORTRAN IV-PLUS
C
C
        CHARACTER C, ALPHABET*26

        DATA ALPHABET/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/

        WRITE(7,99)
99      FORMAT('1 CHARACTER EXAMPLE PROGRAM OUTPUT'/)

        DO 100 I=1,26

        WRITE(7,10) ALPHABET
10      FORMAT(1X, A)

        ALPHABET = ALPHABET(2:) // ALPHABET(1:1)

100     CONTINUE

        CALL REVERSE(ALPHABET)
        WRITE(7,10) ALPHABET

        CALL REVERSE(ALPHABET(1:13))
        WRITE(7,10) ALPHABET

        CALL FIND_SUBSTRINGS('UVW', ALPHABET)
        CALL FIND_SUBSTRINGS('A', 'DAJHDHAJDAHDJA4E CEUEBCUEIAWSAWQLQ')

        WRITE(7,98)
98      FORMAT('0 END OF CHARACTER EXAMPLE PROGRAM OUTPUT')

        STOP
        END

        SUBROUTINE REVERSE(S)
        CHARACTER T, S*(*)

        J = LEN(S)
        IF (J .GT. 1) THEN
            DO 10 I=1, J/2
                T = S(I:I)
                S(I:I) = S(J:J)
                S(J:J) = T
                J = J-1
10              CONTINUE
            ENDIF

        RETURN
        END

        SUBROUTINE FIND_SUBSTRINGS(SUB, S)
        CHARACTER*(*) SUB,S
        CHARACTER*132 MARKS

        I = 1
        K = 1
        MARKS = ' '

10      J = INDEX(S(I:), SUB)
        IF (J .NE. 0) THEN
            I = I + (J-1)
            MARKS(I:I) = '#'
            K = I
            I = I+1
            IF (I .LE. LEN(S)) GO TO 10
            ENDIF

        WRITE(7,100) S, MARKS(:K)
100     FORMAT( 2(/1X, A) )
        END
```

Figure 4-1 Character Data Program Example

```
 CHARACTER EXAMPLE PROGRAM OUTPUT

ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHI
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
RSTUVWXYZABCDEFGHIJKLMNOPQ
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTUV
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
ZYXWVUTSRQPONMLKJIHGFEDCBA
NOPQRSTUVWXYZMLKJIHGFEDCBA

NOPQRSTUVWXYZMLKJIHGFEDCBA
       #

DAJHDHAJDAHDJA4E CEUEBCUEIAWSAWQLQ
 #    #  #   #             #  #

 END OF CHARACTER EXAMPLE PROGRAM OUTPUT
```

Figure 4-2 Output Generated by Example Program

4.8 CHARACTER LIBRARY FUNCTIONS

The VAX-11 FORTRAN IV-PLUS system provides four character functions:

- CHAR
- ICHAR
- INDEX
- LEN

4.8.1 CHAR Function

The CHAR function returns a 1-byte character value equivalent to the integer ASCII value passed as its argument. It has the form:

```
CHAR(i)
```

i

An integer expression equivalent to an ASCII code.

4.8.2 ICHAR Function

The ICHAR function returns an integer ASCII code equivalent to the character expression passed as its argument. It has the form:

```
ICHAR(c)
```

c

A character expression. If c is longer than one byte, the ASCII code equivalent to the first byte is returned, and the remaining bytes are ignored.

4.8.3 INDEX Function

The INDEX function is used to determine the starting position of a substring: it has the form:

```
INDEX(c1,c2)
```

c1

A character expression that specifies the string that is to be searched for a match with the value of c2.

c2

A character expression representing the substring for which a match is desired.

If INDEX finds an instance of the specified substring (c2), it returns an integer value corresponding to the starting location in the string (c1). For example, if the substring sought is CAT, and the string that is searched contains DOGCATFISH, the return value of INDEX is 4.

If INDEX cannot find the specified substring, it returns the value 0.

4.8.4 LEN Function

The LEN function returns an integer value that indicates the length of a character expression. It has the form:

```
LEN(c)
```

c

A character expression

4.9 CHARACTER I/O

The character data type simplifies the transmission of alphanumeric data. You can read and write character strings of any length from 1 to 32767 characters. For example:

```
     CHARACTER*24    TITLE
          .
          .
          .
     READ(12,100)    TITLE
 100 FORMAT(A)
```

These statements cause 24 characters read from logical unit 12 to be stored in the 24-byte character variable TITLE. Compare this with the code necessary if you used Hollerith data stored in numeric variables or arrays:

```
      INTEGER*4  TITLE(6)
           .
           .
           .
      READ(12,100)  TITLE
  100 FORMAT (6A4)
```

Note that you must divide the data into lengths suitable for real or (in this case) integer data, and specify I/O and FORMAT statements to match. In this example, a 1-dimensional array comprising six 4-byte elements is filled with 24 characters from logical unit 12.

CHAPTER 5

FORTRAN CALL CONVENTIONS

VAX-11 FORTRAN IV-PLUS provides a mechanism you can use to gain access to services external to your FORTRAN programs. By including CALL statements or function references in your source program, you can use procedures such as mathematical functions, VAX/VMS system services, and procedures written in languages other than FORTRAN.

Refer to the VAX-11 Common Run-Time Procedure Library Reference Manual for descriptions of the procedures included in the Run-Time Library, and information on how they are called.

5.1 PROCEDURE CALLS

A procedure is a program, such as a FORTRAN function or subroutine, that performs one or more computations for other programs. In many cases, procedures perform calculations that are used widely and repeatedly in many FORTRAN applications. It is more efficient to write these procedures (subprograms) once, and make them available to other programs, than to reinvent them every time the need arises.

Subprograms can be either functions or subroutines. A function is a subprogram that returns a value to the calling program, by assigning the value to the function's name. A subroutine may return values, but a value is not associated with the subroutine name. Functions and subroutines can return values by storing values in elements of the argument list, or in COMMON blocks. See the VAX-11 FORTRAN IV-PLUS Language Reference Manual for information on defining and invoking subprograms.

5.2 VAX-11 PROCEDURE CALLING STANDARD

Programs compiled by the VAX-11 FORTRAN IV-PLUS compiler conform to the standard defined for VAX-11 procedure calls (see Appendix C of the VAX-11 Architecture Handbook, Vol. 1). This standard prescribes how arguments are passed, how function values are returned, and how procedures (such as subprograms) receive and return control. VAX-11 FORTRAN IV-PLUS also provides features that allow FORTRAN programs to call system services and procedures written in other native-mode languages supported in VAX/VMS.

The information in this section pertains to calling system service routines from FORTRAN. If you want to write routines that can be called from FORTRAN programs, you should pay particular attention to the argument list descriptions and to the machine code format description in Section 5.4.

5.2.1 Argument Lists

The VAX-11 procedure calling standard defines an argument list as a sequence of longword (4-byte) entries, the first of which is an argument count. The argument count is the first byte in the first entry in the list. It indicates how many arguments follow in the list.

Memory for FORTRAN argument lists and VAX-11 standard descriptors is usually allocated statically. To optimize space and time, the argument lists are pooled and argument list entries are initialized at compile time, when possible. Sometimes several calls can use the same argument list. For example:

```
X = DIZ(Y)
Z = DIZ(Y) * DIZ(Z)
A = X + Z + DIZ(Y)
```

All of these DIZ(Y) references will use the same argument list.

Omitted arguments (for example CALL X(A,,B)) are represented by an argument list entry that has a value of 0.

See Section 5.4.1 for examples of machine code generated for FORTRAN argument lists.

5.2.2 Argument Passing Mechanisms

The VAX-11 procedure calling standard defines three mechanisms by which arguments are passed to procedures:

1. Call-by-value - the argument list entry is the value
2. Call-by-reference - the argument list entry is the address of the value
3. Call-by-descriptor - the argument list entry is the address of a descriptor of the value

By default, VAX-11 FORTRAN IV-PLUS uses the call-by-reference and call-by-descriptor mechanisms. The call-by-reference mechanism is used to pass all numeric actual arguments: logical, integer, real, double precision, and complex. The call-by-descriptor mechanism is used to pass all character actual arguments.

5.2.3 Argument List Built-In Functions

By default, FORTRAN uses the call-by-reference or call-by-descriptor mechanism for passing arguments, depending on the argument's data type. In some cases, however, a function reference or call to a non-FORTRAN IV-PLUS procedure may require that you supply arguments in a different form. Calls to VAX/VMS system services are such a case. Therefore, FORTRAN provides three built-in functions for passing arguments when you cannot use the FORTRAN default mechanism. These built-in functions are:

```
%VAL
%REF
%DESCR
```

These functions can appear only in actual argument lists.

The following sections describe the use of these functions. Note that the argument list built-in functions are never used to call a procedure written in FORTRAN.

5.2.3.1 **%VAL** - This function forces the argument list entry to use the call-by-value mechanism. It has the form:

```
%VAL(arg)
```

The argument list entry (arg) is the value of the entry. Because argument list entries are longwords, the argument value must be an integer, logical, or real constant, variable, array element, or expression.

5.2.3.2 **%REF** - This function forces the argument list entry to use the call-by-reference mechanism. It has the form:

```
%REF(arg)
```

The argument list entry (arg) is the address of the value. The argument value can be a numeric or character expression, array, array element, or procedure name. This is the default FORTRAN method for passing all numeric values.

5.2.3.3 **%DESCR** - This function forces the argument list entry to use the call-by-descriptor mechanism. It has the form:

```
%DESCR(arg)
```

The argument list entry (arg) is the address of a descriptor of the value. The argument value can be any type of FORTRAN expression. The compiler can generate VAX-11 descriptors for all FORTRAN data types.

Call-by-descriptor is the default FORTRAN mechanism for passing character arguments, because the subprogram may need to know the length of the character argument. In particular, FORTRAN always generates code to refer to character dummy arguments through the addresses in their descriptors.

5.2.3.4 **Examples of %VAL, %REF, %DESCR** - The following examples illustrate the use of the argument list built-in functions.

```
CALL SUB(2,%VAL(2))
```

The first constant is passed by reference. The second constant is passed by value.

```
CHARACTER*10 A,B
CALL SUB(A,%REF(B))
```

The first character variable is passed by descriptor. The second character variable is passed by reference.

```
INTEGER IARY(20), JARY(20)
CALL SUB(IARY,%DESCR(JARY))
```

The first array is passed by reference. The second array is passed by descriptor.

See Section 5.4.2 for examples that include the generated machine code.

5.2.4 Function Return Values

The method of returning function procedure values depends on the data type of the value, as summarized in Table 5-1.

Table 5-1
Function Return Values

Data Type	Return Method
Logical Integer Real	General register R0
Double Precision	R0: High-order result R1: Low-order result
Complex	R0: Real part R1: Imaginary part
Character	An extra entry is added as the first entry of the argument list. This new first argument entry points to a character string descriptor. At run time, storage is allocated to contain the value of the result, and the proper address is stored in the descriptor.

5.2.5 %LOC Built-In Function

The %LOC built-in function computes the address of a storage element, as an INTEGER*4 value. This value can be used in an arithmetic expression. This has particular applicability for certain system services or non-FORTRAN procedures that may require arguments that contain the addresses of storage elements.

5.3 CALLING VAX/VMS SYSTEM SERVICES

The VAX/VMS operating system provides a number of service procedures you can call from FORTRAN programs. The following subsections describe the methods you can use to call system service procedures, and to ensure that you pass and receive information in the correct form.

You can invoke system services from a FORTRAN program by including a function reference or a subroutine CALL statement in your program, specifying the system service you want to use. To specify a system service, use the form:

```
SYS$service-name (a,...,a)
```

You pass arguments to the system services according to the requirements of the particular service you are calling: a value, an address, or the address of a descriptor may be needed, as described in Section 5.3.3, below. See the VAX/VMS System Services Reference Manual for a full definition of all services.

5.3.1 Calling System Services by Function Reference

In most cases, you should check the return status after calling a system service. Therefore, you should call system services by function reference rather than by issuing a subroutine CALL.

NOTE

System service functions must be declared as INTEGER*4 functions.

For example:

```
INTEGER*4  SYS$CREMBX
INTEGER*2  ICHAN
MBX = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
IF (.NOT. MBX) GO TO 100
```

In this example, the system service referred to is the Create Mailbox service. First, the system service name is declared, then an INTEGER*2 variable (ICHAN) is declared, to receive the channel number.

The function reference allows a return status value to be stored in the variable MBX, which can then be checked for .TRUE. or .FALSE. on return. If the function's return status is .FALSE., indicating failure, control transfers to statement 100, at which point some form of error processing can be undertaken. You can also check for a particular return status, such as an access violation, by comparing the return status to one of the status codes defined by the system. For example:

```
IF (MBX .EQ.  SS$_ACCVIO) THEN
```

Refer to the VAX/VMS System Services Reference Manual for information concerning return status codes. The return status codes are included in the description of each system service.

5.3.2 Calling System Services as Subroutines

Subroutine calls to system services are made in the same way that calls are made to any other subroutine. For example, to call the Create Mailbox system service, issue a CALL to SYS$CREMBX, passing to it the appropriate arguments, as:

```
CALL SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
```

This CALL corresponds to the function reference shown above. The main difference is that the status code returned by the system service is not tested. For this reason, you should avoid this method of calling system services.

5.3.3 Passing Arguments to System Services

Generally, system services require input arguments to be passed by value, and output arguments to be passed by reference. Therefore, you must use the %VAL built-in function when passing input arguments to certain system services, to make sure they are passed the correct data. Some system services require character arguments (see Section 5.3.3.3). To determine the argument requirements for a system service, see the VAX/VMS System Services Reference Manual.

5.3.3.1 **Input and Output Address Arguments** - You will often need to tell the system service where to find input values and where to store output values. Thus, you must determine the hardware data type of the argument: byte, word, longword, or quadword.

For input arguments that refer to byte, word, or longword values, you can specify either constants or variables. If you specify a variable, you must declare it to be equal to or longer than the data type required. Table 5-2 lists the variable data type requirements.

For output arguments you must declare a variable of exactly the length required, to avoid including extraneous data. If, for example, the system returns a byte value in a word-length variable, the leftmost eight bits of the variable will not be overwritten on output; thus, the variable will not contain the data you expect.

Table 5-2
Variable Data Type Requirements

VAX/VMS Type Required	Input Argument Declaration	Output Argument Declaration
Byte	BYTE, INTEGER*2, INTEGER*4	BYTE
Word	INTEGER*2, INTEGER*4	INTEGER*2
Longword	INTEGER*4	INTEGER*4
Quadword	Properly dimensioned array	Properly dimensioned array
Indicator	LOGICAL	

For arguments described as "address of an entry mask" or "address of a routine," declare the argument value as an external procedure. For example, if a system service requires the address of a routine and you want to specify the routine PROGA, specify

```
EXTERNAL PROGA
```

in the declarations portion of the FORTRAN program to define the address of the routine for use as an input argument.

To store output produced by system services, you must allocate sufficient space to contain the output. You do this by declaring variables of the proper size. For example, the Create Mailbox system service produces a two-byte value. Thus, you set up space as follows:

```
INTEGER*2  ICHAN
INTEGER*4  SYS$CREMBX
   .
   .
   .
MBX = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
```

If the output is a quadword value, you must declare an array of the proper dimensions. For example, the Get Time system service (SYS$GETTIM) returns the time as a quadword binary value. Thus, you would need to specify the following:

```
INTEGER*4  SYSTIM(2)        ! DECLARE ARRAY
INTEGER*4  SYS$GETTIM
   .
   .
   .
ISTAT = SYS$GETTIM(SYSTIM) ! GET TIME
```

The type declaration INTEGER*4 SYSTIM(2) sets up a vector consisting of two longwords, into which the time value will be stored.

5.3.3.2 **Defaults for Optional Arguments** - All optional arguments have default values. You must use commas in place of omitted arguments. For example, the Translate Logical Name (SYS$TRNLOG) system service takes five arguments. If you omit the last two arguments you must include two commas in their place, as follows:

```
ISTAT = SYS$TRNLOG('LOGNAM',LENGTH,BUFFA,,)
```

An invalid reference would result if you specified:

```
ISTAT = SYS$TRNLOG('LOGNAM',LENGTH,BUFFA)
```

This reference provides only three arguments, not the requisite five.

5.3.3.3 **Passing Character Arguments** - Some VAX/VMS system services (for example, the Translate Logical Name system service) require character arguments for either input or output. The Translate Logical Name system service (SYS$TRNLOG) accepts a logical name as input, and returns the associated logical name or file specification, if any, as output. VAX/VMS system services that process character data require that arguments be passed by descriptor. FORTRAN passes all character data by descriptor. On input, a character constant, variable, array element, or expression is passed to the system service by descriptor. On output, two arguments are needed: the character variable or array element to hold the output string, and an INTEGER*2 variable, which is set to the actual length of the output string. For example:

```
CHARACTER*64 BUFFA
INTEGER*2 LENGTH
INTEGER*4 SYS$TRNLOG
   .
   .
   .
ISTAT = SYS$TRNLOG('LOGNAM',LENGTH,BUFFA,,)
```

The logical name LOGNAM is translated to its associated name or file specification, and the output values, length and associated name or file specification, are stored in the locations you specified -- LENGTH and BUFFA, respectively.

5.4 MACHINE CODE EXAMPLES

The following sections present examples of FORTRAN calls and their corresponding machine code.

5.4.1 Argument Passing Examples

The format used in the following examples shows FORTRAN source lines, followed by generated object code argument lists.

Example 1:

FORTRAN Source Code

```
REAL X
INTEGER J(10)
CHARACTER*15 C
CALL SUB (X,J(3),C)
```

Object Code

```
ARGLST:   .LONG 3          ; COUNT
          .ADDR X          ; ADDRESS OF X
          .ADDR J+8        ; ADDRESS OF J(3)
          .ADDR L$1        ; C DESCRIPTOR ADDRESS

L$1:      .WORD 15         ; LENGTH OF C
          .BYTE 14         ; CHARACTER TYPE CODE
          .BYTE 1          ; SCALAR CLASS CODE
          .ADDR C          ; ADDRESS OF C
```

This example shows how the compiler generates an argument list for the arguments specified in the CALL statement. The compiler can initialize the addresses of real variable X and array element J(3) because these are explicitly specified in the CALL statement. Similarly, the compiler has enough information to generate an initialized descriptor for the character string C.

Example 2:

FORTRAN Source Code

```
REAL X(10)
CHARACTER*15 C
CALL SUB (X(I), C(J:K))
```

Object Code

```
ARGLST:   .LONG 2           ; COUNT
          .LONG 0           ; X(I) INITIALIZED AT RUN TIME
          .ADDR L$1         ; C(J:K) DESCRIPTOR ADDRESS

L$1:      .WORD 0           ; C(J:K) LENGTH, SET AT RUN TIME
          .BYTE 14          ; CHARACTER TYPE CODE
          .BYTE 1           ; SCALAR CLASS CODE
          .LONG 0           ; BASE ADDRESS OF C(J:K), SET AT RUN
                            ; TIME
```

Run-time argument list initialization code

```
MOVL      I,R0              ; COMPUTE ADDRESS OF X(I) AND
MOVAF     X-4[R0],ARGLST+4  ; STORE IT IN THE ARGUMENT LIST
SUBL3     #1,J,R0           ; COMPUTE THE LENGTH OF C(J:K)
SUBL3     R0,K,R1           ;
MOVW      R1,L$1            ; STORE LENGTH OF C(J:K) IN ARGUMENT
                            ; LIST
MOVAB     C[R0],L$1+4       ; STORE BASE ADDRESS OF C(J:K) IN
                            ; ARGUMENT LIST
CALLG     ARGLST,SUB        ; CALL SUBROUTINE SUB
```

In this example, the FORTRAN source lines define a real array X, comprising 10 elements; and a character variable C, comprising 15 elements. The actual arguments passed to subroutine SUB are the I-th element of array X, and the substring of C from the J-th to the K-th character. The compiler generates an argument list consisting of three longwords; the first is the count, and the next two are (respectively) the address of the I-th element of X and the address of the descriptor of substring C(J:K). Note that the address of X(I) and C(J:K), and the length of C(J:K) are initialized to 0, because at compile time these values are unknown.

5.4.2 Argument List Built-In Function Examples

In the following examples, the FORTRAN source code is shown first, followed by the generated object code.

Example 1: %VAL

FORTRAN Source Code

```
CALL SUB (4, %VAL(6))
```

Object Code

```
ARGLST:   .LONG 2          ; COUNT
          .ADDR CON4       ; ADDRESS OF CONSTANT
          .LONG 6          ; VALUE

CON4:     .LONG 4          ;
```

As shown, the compiler generates an address for the constant 4 in the first entry, but generates the actual value (6) in the following entry.

Example 2: %REF

FORTRAN Source Code

```
CHARACTER*10 C,D
CALL SUB (C, %REF(D))
```

Object Code

```
ARGLST:   .LONG 2          ; COUNT
          .ADDR L$1        ; ADDRESS OF C DESCRIPTOR
          .ADDR D          ; ADDRESS OF D

L$1:      .WORD 10         ; LENGTH
          .BYTE 14         ; TYPE CODE
          .BYTE 1          ; CLASS CODE
          .ADDR C          ; ADDRESS
```

As shown, the argument list entry for D is the address of D. The compiler does not generate a descriptor for D, as it does for C, even though C and D are both specified in the source program as character variables.

Example 3: %DESCR

FORTRAN Source Code

```
CALL SUB (X, %DESCR(X))
```

Object Code

```
ARGLST:   .LONG 2          ; COUNT
          .ADDR X          ; ADDRESS OF X
          .ADDR L$1        ; ADDRESS OF X DESCRIPTOR

L$1:      .WORD 4          ; LENGTH
          .BYTE 10         ; TYPE CODE
          .BYTE 1          ; CLASS CODE
          .ADDR X          ; ADDRESS
```

In this example, the first argument list entry contains an address, while the second entry contains a pointer to a descriptor.

5.4.3 Character Functions

The following example illustrates how character function argument lists are generated.

FORTRAN Source Code

```
CHARACTER*10 C,D
D = C(I,J)
```

Object Code

```
ARGLST:   .LONG 3          ; COUNT
          .ADDR L$1        ; ADDRESS OF FUNCTION DESCRIPTOR
          .ADDR I          ; ADDRESS OF I
          .ADDR J          ; ADDRESS OF J

L$1:      .WORD 10         ; LENGTH
          .BYTE 14         ; TYPE CODE
          .BYTE 1          ; CLASS CODE
          .LONG 0          ; ADDRESS

SUBL2     #10,SP           ; ALLOCATE SPACE FOR 10 CHARACTERS
MOVL      SP,L$1+4         ; SET ADDRESS
CALLG     ARGLST,C         ; CALL FUNCTION C
MOVC3     #10,(SP),D       ; MOVE RESULT TO D
MOVL      R1,SP            ; REMOVE RESULT FROM STACK
```

In this example, an additional argument list entry is allocated; the descriptor of the return value of the character function C.

CHAPTER 6

ERROR PROCESSING AND CONDITION HANDLERS

During program execution, your program may occasionally encounter errors or exception conditions. These conditions may result from errors that occur during I/O operations, invalid input data, argument errors in calls to the mathematical library, arithmetic errors, or system-detected errors. VAX-11 FORTRAN IV-PLUS provides three methods of controlling and recovering from errors:

1. Run-Time Library default error-processing procedures

2. ERR= and END= specifications in I/O statements

3. VAX-11 Condition Handling Facility (including user-written condition handlers)

These error-processing methods are complementary, and can be used together in the same program. Thus, you have a number of options to choose from in dealing with errors. You can, of course, allow the Run-Time Library to handle errors for you. This is the default case. Or, you can use ERR= and/or END= specifications in I/O statements to provide special processing if an error or end-of-file occurs while the I/O statement is being executed. Or, you can provide condition handlers to tailor error processing to the special requirements of your applications. Note: This option should be undertaken only by more experienced users, in particular those familiar with the VAX-11 Condition Handling Facility.

The Run-Time Library provides default processing for all exception conditions that occur during FORTRAN program execution. It generates appropriate messages, and takes action to recover from errors where possible. Section 6.1 describes FORTRAN-specific Run-Time Library error processing. Appendix B describes FORTRAN-specific Run-Time Library error messages.

The VAX-11 Condition Handling Facility provides all error processing in the Run-Time Library. You can also use it directly to provide procedures (user-written condition handlers) for processing exception conditions that occur during your program's execution. For example, you can include code to respond to certain kinds of errors in a way that is more appropriate to a particular application than the processing performed by the Run-Time Library. The use of condition handlers requires considerable experience: it is not appropriate for novice users. You should be familiar with the condition handling description in the VAX-11 Common Run-Time Procedure Library Reference Manual, and in the VAX-11/780 Architecture Handbook before you attempt to write a condition handler.

6.1 RUN-TIME LIBRARY DEFAULT ERROR PROCESSING

The Run-Time Library contains condition handlers that process a number of errors that may occur during FORTRAN program execution. A default action is defined for each FORTRAN-specific exception condition that the Run-Time Library recognizes; Table 6-1 lists these actions. These default actions occur unless overridden by a user-written condition handler (see Section 6.2).

Some errors result in recovery action. For example, you can use the ERR= specification to code an error statement label in an I/O statement. In some cases, the error may be ignored and execution will continue. Or, recovery may be initiated based on the nature of the error. You can use the ERRSNS system subroutine to determine which error occurred. ERRSNS returns a FORTRAN-specific error number and system status codes for the last error that occurred; see Table 6-1. The ERRSNS routine is described in Appendix C.

6.1.1 Using ERR= and END= Transfers

By including an ERR=label or END=label specification in an I/O statement, you can transfer control to error processing code in your program. If you use an END= or ERR= transfer to process an I/O error, no error message is printed and execution continues at the designated statement. However, if a severe error occurs while an I/O statement is being processed, and you did not specify an ERR= transfer, the default action is to print an error message and terminate execution.

If you specify ERR=label in an I/O statement and an error occurs during I/O statement execution, the Run-Time Library transfers program control to the statement at the label specified. For example:

```
WRITE(8,50,ERR=400)
```

If an error occurs during the write operation, control transfers to the statement at label 400.

You can also specify ERR=label as a keyword in an OPEN or CLOSE statement. For example:

```
OPEN(UNIT=INFILE,TYPE='OLD',ERR=100,NAME=FILN)
```

If errors are detected during OPEN statement execution, control transfers to statement 100.

The END=label specification can be used to handle an end-of-file condition, as follows:

```
READ(12,70,END=550)
```

If an end-of-file is detected while this I/O statement is being executed, control transfers to statement 550.

If an end-of-file is detected while a READ statement is being executed, and you did not specify END=label, an error condition occurs. If you specified ERR=label, control is transferred to the specified statement.

6.1.2 Run-Time Library Error Processing Control

The Run-Time Library's error processing depends on three factors: the severity of the error; whether the error permits continuation; and whether an ERR= transfer was provided in the case of an I/O error.

Table 6-1 lists the FORTRAN-specific errors processed by the Run-Time Library. For each error, the table shows the error message text, the symbolic condition name, the FORTRAN-specific error code, the severity code of the error, and the type of recovery action.

A severity code is included as part of the error code that is generated when an error condition is detected. All FORTRAN-specific errors have severity codes of either error (E) or severe error (F). As shown in Table 6-1, most FORTRAN-specific errors are severe. If no recovery action is specified for a severe error, program execution terminates by default.

Two types of recovery are possible: ERR= and RETURN. Most I/O errors include ERR= recovery. However, if you specified no ERR= transfer, and the error severity was F, your program terminates with an error message, and an exit status of severe error. If you did specify an ERR= transfer, the message will not be displayed. Thus, if you receive one of the severe error messages, the implication is that no ERR= transfer was specified. You can circumvent the error, and the resulting program termination, by including an ERR= transfer for this error in your source program.

A recovery type of RET means that recovery and continuation are possible only if you establish a condition handler for that error. User-written condition handlers are described in Section 6.3.

The letter C in the "Sev" column means that you can continue execution immediately after the error if a user-written condition handler has changed the severity code to error (E) or warning (W). If there is no letter C in the "Sev" column, you cannot continue execution immediately after the error. If you attempt to do so, program execution will be terminated.

When errors occur for which no recovery type is specified, the program exits; that is, execution of the program is terminated, and an error message is printed. To prevent program termination, include a condition handler that performs an unwind (see Section 6.3).

Table 6-1
Summary of FORTRAN Run-Time Errors

FORTRAN Condition Symbol	Err #	Sev	Rec. Type	Message Text
FOR$_NOTFORSPE	1	F		NOT A FORTRAN-SPECIFIC ERROR
FOR$_REWERR	20	F	ERR=	REWIND ERROR
FOR$_DUPFILSPE	21	F	ERR=	DUPLICATE FILE SPECIFICATIONS
FOR$_INPRECTOO	22	F	ERR=	INPUT RECORD TOO LONG
FOR$_BACERR	23	F	ERR=	BACKSPACE ERROR
FOR$_ENDDURREA	24	F	ERR=	END-OF-FILE DURING READ
FOR$_RECNUMOUT	25	F	ERR=	RECORD NUMBER OUTSIDE RANGE
FOR$_OPEDEFREQ	26	F	ERR=	OPEN OR DEFINEFILE REQUIRED...
FOR$_MORONEREC	27	F	ERR=	MORE THAN ONE RECORD IN I/O STATEMENT
FOR$_CLOERR	28	F	ERR=	CLOSE ERROR
FOR$_FILNOTFOU	29	F	ERR=	FILE NOT FOUND
FOR$_OPEFAI	30	F	ERR=	OPEN FAILURE
FOR$_MIXFILACC	31	F	ERR=	MIXED FILE ACCESS MODES
FOR$_INVLOGUNI	32	F	ERR=	INVALID LOGICAL UNIT NUMBER
FOR$_ENDFILERR	33	F	ERR=	ENDFILE ERROR
FOR$_UNIALROPE	34	F	ERR=	UNIT ALREADY OPEN
FOR$_SEGRECFOR	35	F	ERR=	SEGMENTED RECORD FORMAT ERROR
FOR$_ATTREANON	36	F	ERR=	ATTEMPT TO READ NON-EXISTENT RECORD
FOR$_INCRECLEN	37	F	ERR=	INCONSISTENT RECORD LENGTH
FOR$_ERRDURWRI	38	F	ERR=	ERROR DURING WRITE
FOR$_ERRDURREA	39	F	ERR=	ERROR DURING READ
FOR$_RECIO_OPE	40	F	ERR=	RECURSIVE I/O OPERATION
FOR$_INSVIRMEM	41	F	ERR=	INSUFFICIENT VIRTUAL MEMORY
FOR$_NO_SUCDEV	42	F	ERR=	NO SUCH DEVICE
FOR$_FILNAMSPE	43	F	ERR=	FILE NAME SPECIFICATION ERROR
FOR$_RECSPEERR	44	F	ERR=	RECORD SPECIFICATION ERROR
FOR$_KEYVALERR	45	F	ERR=	KEYWORD VALUE ERROR IN OPEN STATEMENT
FOR$_INCOPECLO	46	F	ERR=	INCONSISTENT OPEN/CLOSE PARAMETERS
FOR$_WRIREAFIL	47	F	ERR=	WRITE TO READONLY FILE
FOR$_INVARGFOR	48	F	ERR=	INVALID ARGUMENT TO FORTRAN I/O LIBRARY
FOR$_LISIO_SYN	59	F,C	ERR=	LIST-DIRECTED I/O SYNTAX ERROR
FOR$_INFFORLOO	60	F	ERR=	INFINITE FORMAT LOOP
FOR$_FORVARMIS	61	F,C	ERR=	FORMAT/VARIABLE-TYPE MISMATCH
FOR$_SYNERRFOR	62	F	ERR=	SYNTAX ERROR IN FORMAT
FOR$_OUTCONERR	63	E,C	ERR=	OUTPUT CONVERSION ERROR
FOR$_INPCONERR	64	F,C	ERR=	INPUT CONVERSION ERROR
FOR$_OUTSTAOVE	66	F	ERR=	OUTPUT STATEMENT OVERFLOWS RECORD
FOR$_INPSTAREQ	67	F	ERR=	INPUT STATEMENT REQUIRES TOO MUCH DATA
FOR$_VFEVALERR	68	F,C	ERR=	VARIABLE FORMAT EXPRESSION VALUE ERROR
SS$_INTOVF	70	F,C	RET	INTEGER OVERFLOW
SS$_INTDIV	71	F,C	RET	INTEGER ZERO DIVIDE
SS$_FLTOVF	72	F,C	RET	FLOATING OVERFLOW
SS$_FLTDIV	73	F,C	RET	FLOATING ZERO DIVIDE
SS$_FLTUND	74	F,C	RET	FLOATING UNDERFLOW
SS$_SUBRNG	77	F,C	RET	SUBSCRIPT OUT OF RANGE
MTH$_WRONUMARG	80	F		WRONG NUMBER OF ARGUMENTS
MTH$_INVARGMAT	81	F		INVALID ARGUMENT TO MATH LIBRARY
MTH$_UNDEXP	82	F,C	RET	UNDEFINED EXPONENTIATION
MTH$_LOGZERNEG	83	F,C	RET	LOGARITHM OF ZERO OR NEGATIVE VALUE
MTH$_SQUROONEG	84	F,C	RET	SQUARE ROOT OF NEGATIVE VALUE
MTH$_SINCOSSIG	87	F,C	RET	SINE OR COSINE SIGNIFICANCE LOST
MTH$_FLOOVEMAT	88	F,C	RET	FLOATING OVERFLOW IN MATH LIBRARY
MTH$_FLOUNDMAT	89	F,C	RET	FLOATING UNDERFLOW IN MATH LIBRARY
FOR$_ADJARRDIM	93	F		ADJUSTABLE ARRAY DIMENSION ERROR

NOTES:

1. The ERR= transfer is taken after completion of the I/O statement for continuable errors numbered 59, 61, 63, 64, and 68; the resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR= transfer as soon as the error is detected; thus, file status and record position are undefined.

2. If no ERR= address has been defined for error 63, the program continues after the error message is printed. The entire overflowed field is filled with asterisks, to indicate the error in the output record.

3. Function return values for errors numbered 82, 83, 84, 87, 88, and 89 can be modified by means of user-written condition handlers. See Section 6.4.

4. Error number 1 (FOR$_NOTFORSPE) indicates that an error was detected that was not a FORTRAN-specific error; that is, it was not reportable through any other message in the table. If you call ERRSNS, an error of this kind returns a value of 1. To obtain the unique system condition value that identifies the error, use the fifth argument of the call to ERRSNS (condval). See Section C.4.

6.1.3 Using the ERRSNS Subroutine

You can use the ERRSNS system subroutine to process errors. ERRSNS lets you determine the exact nature of the error and take action accordingly. Table 6-1 contains a FORTRAN-specific integer error number for each of the errors in the table. This error number can be obtained by calling ERRSNS after an error occurs. In many situations you can provide code to react to specific I/O errors, once you know which error occurred. To get this information, use the ERRSNS routine, as shown in the following example.

```
        CHARACTER*40 FILN
10      ACCEPT *, FILN
        OPEN(UNIT=INFILE,TYPE='OLD',ERR=100,NAME=FILN)
                .
                .    (process input file)
                .
100     CALL ERRSNS (IERR)
        IF (IERR .EQ. 29) THEN
           TYPE *, 'FILE:', FILN, 'DOES NOT EXIST, ENTER NEW FILENAME'
        ELSE IF (IERR .EQ. 43) THEN
           TYPE *, 'FILENAME:', FILN, 'WAS BAD, ENTER NEW FILENAME'
        ELSE
           TYPE *, 'FAILURE ON INPUT FILE, I/O ERROR CODE=', IERR
           STOP
        ENDIF
        GO TO 10
        END
```

As shown, the OPEN statement contains an ERR=100 keyword, causing a branch to the ERRSNS subroutine if an error occurs during execution of the OPEN. The ERRSNS subroutine returns an error number value in the integer variable IERR. The program then uses the value of IERR either to print a message indicating the nature of the error and continue, or to print a message indicating that the error was too severe to allow the program to continue.

See Appendix C for more information about the ERRSNS subroutine.

6.2 OVERVIEW OF THE VAX-11 CONDITION HANDLING FACILITY

A condition handler is a procedure that is invoked when an exception condition occurs. The exception condition may be either hardware or software related. Hardware exceptions include floating overflows, memory access violations, and the use of reserved operands. Software exceptions include output conversion errors, end-of-file conditions, and invalid arguments to mathematical procedures.

When the VAX/VMS system creates a user process, it establishes a system-defined condition handler that will, in the absence of any user-written condition handler, process errors that occur during execution of the user image. Thus, by default, run-time errors will cause this condition handler to print one of the standard error messages and terminate or continue execution, depending on the severity code associated with the error.

This default condition handler is the last condition handler reached when a search is undertaken to find a condition handler to respond to an error. If no lower-level procedure has established a handler, or if all lower-level handlers have resignaled (see the list of definitions below), the default condition handler responds to the error. The default handler calls the system's message output routine, to send the appropriate message to the user. Messages are sent to the SYS$OUTPUT file, and to the SYS$ERROR file, if these files are both present. If the condition was one that permits continuation, program execution continues. Otherwise, the default handler forces program termination; and the condition value (see Table 6-1) becomes the program exit status.

You can create and establish your own condition handlers to accommodate the needs of your applications. For example, you can create and display messages that specifically describe conditions encountered during execution of an application program, instead of relying on the standard system error messages.

6.2.1 Definitions

Before reading further, you should become familiar with the following terms:

- **Condition value** - an INTEGER*4 value that identifies a particular exception condition. See Section 6.3.4.
- **Procedure** - an executable program unit; a main program, subroutine, or function.
- **Procedure activation** - the environment in which a procedure executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the procedure activation. A new procedure activation is created every time a procedure is called; the procedure activation is deleted when the procedure returns.
- **Condition handler** - a function that a procedure activation specifies as the procedure to be called when an exception condition occurs.

- **Establish** - the process of placing the address of a condition handler in the stack frame for the current procedure activation. A condition handler established for a procedure is called automatically when an exception condition occurs. In FORTRAN, condition handlers are established by means of the LIB$ESTABLISH procedure. See Section 6.4.2.

- **Signal** - the means by which the occurrence of an exception condition is made known; signals are initiated by a call to a signal procedure, which then calls a condition handler. There are two signal procedures:

  LIB$SIGNAL - signal a condition and possibly continue program execution;

  LIB$STOP - signal a severe error and do not continue program execution, unless a condition handler performs an unwind.

  See Section 6.2.2.

- **Resignal** - the means by which a condition handler indicates that the signal procedure is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the SS$_RESIGNAL value. See Section 6.3.3.

- **Condition symbol** - a symbol used to specify a condition value, in the form:

  fac$_symbol

  where **fac** is a facility name prefix and **symbol** identifies a specific condition.

  Table 6-1 lists FORTRAN-related condition symbols.

- **Unwind** - the act of returning control to a particular procedure activation, bypassing any intermediate procedure activations. For example, if X calls Y and Y calls Z; and Z detects an error; then a condition handler associated with Z can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

6.2.2 Condition Signals

A condition signal consists of a call to one of the two system-supplied signal procedures, in either of the following forms:

```
CALL LIB$SIGNAL (condition-value, arg,...)

CALL LIB$STOP (condition-value, arg,...)
```

You can use a condition signal when you do not want to handle a condition in the routine in which it was detected, but wish to pass the information to a higher-level routine. If the current procedure can continue after the signal is made, call LIB$SIGNAL. A higher-level procedure can then determine whether program execution is continued. If the condition will not allow the current procedure to continue, call LIB$STOP.

To pass the condition value, you must use the %VAL argument list built-in function (see Section 5.2.3.1). Condition values are usually expressed as condition symbols. For example:

```
CALL LIB$SIGNAL (%VAL(MTH$_FLOOVEMAT))
```

Additional arguments may be included to provide supplementary information about the error.

When called, a signal procedure searches for a condition handler by examining the preceding stack frames in order, until it finds a procedure that handles the condition, or the default condition handler is reached.

6.2.3 Handler Responses

A condition handler responds to an exception by taking action in three major areas: 1) condition correction, 2) condition reporting, and 3) execution control.

First, the handler may determine whether it can correct the condition. If it can, the handler will take the appropriate action, and execution will continue. If it cannot correct the condition, the handler may resignal the condition. That is, it may request that another condition handler process the exception.

Condition reporting performed by handlers can involve one of the following actions:

- Maintaining a count of exceptions encountered during program execution
- Resignaling the same condition to send the appropriate message to your terminal or log file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition, for example, to produce a message oriented to a specific application

Execution can be affected in a number of ways. Among them are:

- Continuing from the signal. If the signal was issued through a call to LIB$STOP, the program will exit.
- Unwinding to the establisher at the point of the call that resulted in the exception. The handler specifies the called procedure's function value to be returned.
- Unwinding to the establisher's caller (the procedure that called the procedure that established the handler). The handler specifies the called procedure's function value to be returned.

6.3 USER-WRITTEN CONDITION HANDLERS

The following sections describe how to code and establish a condition handler, and provide some simple examples. See Appendix C of the VAX-11/780 Architecture Handbook, and the VAX-11 Common Run-Time Procedure Library Reference Manual for more details on condition handlers.

6.3.1 Establishing and Removing Handlers

When a procedure is called, no condition handler is initially established. If you want to use a condition handler, you must establish it by calling the Run-Time Library procedure LIB$ESTABLISH and including in the call such code as the following:

```
EXTERNAL HANDLER
CALL LIB$ESTABLISH (HANDLER)
```

To remove an established handler, call the Run-Time Library procedure LIB$REVERT, as follows:

```
CALL LIB$REVERT
```

As a result of this call, the condition handler established in the current procedure activation is removed. When a procedure returns, the condition handler established by the procedure is automatically removed.

6.3.2 FORTRAN Condition Handlers

A FORTRAN condition handler is an INTEGER*4 function that is called when an exception condition occurs. You must define two dummy arguments for a condition handler:

1. An integer array to refer to the argument list from the call to the signal procedure (the "signal arguments"). That is, the list of arguments included in CALL LIB$SIGNAL or CALL LIB$STOP (see Section 6.2.2).

2. An integer array to refer to information concerning the procedure activation that established the condition handler (the "mechanism arguments").

For example, you can define a condition handler as follows:

```
INTEGER*4  FUNCTION  HANDLER(SIGARGS,MECHARGS)
INTEGER*4  SIGARGS(6),MECHARGS(5)
```

The array SIGARGS is used to obtain information from the signal procedure. The values from the signal procedure are listed below.

Value	Meaning
SIGARGS(1)	Indicates how many arguments are being passed in this vector (argument count).
SIGARGS(2)	Indicates the condition being signaled (condition value). See Section 6.3.4 for a discussion of condition values.
SIGARGS(3 to n)	Indicates optional arguments as specified by the call to LIB$SIGNAL or LIB$STOP; note that the dimension bounds for the SIGARGS array should specify as many entries as necessary to refer to the optional arguments

The array MECHARGS is used to obtain information about the procedure activation status of the procedure that established the condition handler. MECHARGS is a 5-element array in the form:

```
MECHARGS(1)        4
MECHARGS(2)        FRAME
MECHARGS(3)        DEPTH
MECHARGS(4)        R0
MECHARGS(5)        R1
```

The first element (MECHARGS(1)) is the argument count of this vector (4).

FRAME (MECHARGS(2)) contains the address of the procedure activation stack frame that established the handler.

DEPTH (MECHARGS(3)) contains the depth (number of calls) that have been made from the procedure activation, up to the point at which the exception occurred.

MECHARGS(4) and MECHARGS(5) contain the values of registers R0 and R1 at the time of the signal.

6.3.3 Handler Function Return Values

Condition handlers specify function return values to control subsequent execution. Function return values and their effects are defined in Table 6-2.

Table 6-2
Condition Handler Function Return Values

Value	Effect
SS$_CONTINUE	Continue execution from the signal. If the signal was issued by a call to LIB$STOP, however, the program exits.
SS$_RESIGNAL	Resignal, to continue the search for a condition handler to process the condition.

A condition handler can also request a stack unwind, by calling SYS$UNWIND before returning. If SYS$UNWIND is called, the function return value is ignored. The handler modifies the saved registers R0 and R1 in the mechanism arguments to specify the called procedure's function value.

A stack unwind can be made to one of two places:

- Unwind to the establisher, at the point of the call that resulted in the exception. Specify:

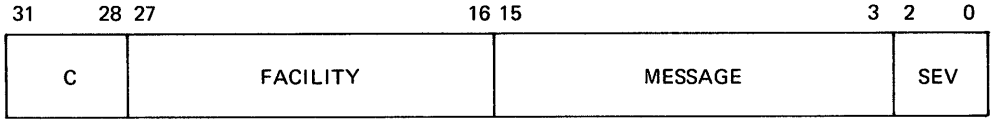
```
CALL SYS$UNWIND(%VAL(MECHARGS(3)),%VAL(0))
```

- Unwind to the procedure that called the establisher. Specify:

```
CALL SYS$UNWIND(%VAL(0),%VAL(0))
```

6.3.4 Condition Values and Symbols

Condition values are used by VAX-11 to indicate that a called procedure has either executed successfully or failed, and to report exception conditions. Condition values are INTEGER*4 values, consisting of fields that indicate which system component generated the value; the reason the value was generated; and the severity of the condition. A condition value has the form:



C field (31:28) - control bits

Facility (27:16) - identifies the software component that generated the condition value. Bit 27 = 1 to indicate a customer facility. Bit 27 = 0 to indicate a DIGITAL facility.

Message (15:3) - describes the condition that occurred. Bit 15 = 1 to indicate the message is specific to a single facility. Bit 15 = 0 to indicate a system wide message.

Sev (2:0) - severity code, as follows:

0 - warning
1 - success
2 - error
3 - information
4 - severe error
5, 6, 7 - reserved

A warning severity code (0) indicates that output was produced, but the results might not be what you expected. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results will not all be correct. A severe error code (4) indicates that the error was of such severity that output was not produced. One of the functions that can be performed by a condition handler is altering the severity code of a condition value, to allow execution to continue or to force an exit, depending on the circumstances.

The condition value is passed as the second element of the array SIGARGS. There are times when you may require that a particular condition be identified by an exact match. That is, each bit of the condition value (31:0) must match the specified condition. For example, you may want to process a floating overflow condition only if its severity code is still 4 (that is, if no previous handler has changed the severity code). As noted above, a typical handler response is to change the severity code and resignal.

In many cases, however, you may want to respond to a condition, regardless of the value of the severity code. To ignore the severity and control fields of a condition value, use the LIB$MATCH_COND function, as follows:

```
index = LIB$MATCH_COND (SIGARGS(2),con-1,con-2,...con-n)
```

This function compares bits 27:3 of the value in SIGARGS(2) with each of the condition values that follow (con-1 through con-n). If it finds a match, the index value is assigned according to the position of the matching condition value in the list. That is, if the match is with the third condition value following SIGARGS(2), then index = 3. If no match is found, index = 0. The value of the index can then be used to transfer control, as in the following example:

```
      INTEGER*4 FUNCTION HANDL(SIGARGS,MECHARGS)
      INCLUDE 'SYS$LIBRARY:FORDEF'
      INTEGER*4 SIGARGS(6),MECHARGS(5)

      INDEX=LIB$MATCH_COND(SIGARGS(2),FOR$_FILNOTFOU,
     1    FOR$_NO_SUCDEV,FOR$_FILNAMSPE,FOR$_OPEFAI)

      GO TO (100,200,300,400),INDEX
      HANDL=SS$_RESIGNAL
      RETURN
        .
        .
        .
```

If no match is found between the condition value in SIGARGS(2) and any of the values in the list, INDEX=0, and control is transferred to the next executable statement after the computed GO TO. A match with any of the values in the list transfers control to the corresponding statement in the GO TO list. Thus, if SIGARGS(2) matches FOR$_OPEFAI, control is transferred to statement 400. Note the use of condition symbols to represent condition values. Table 6-1 lists the FORTRAN condition symbols and their values.

The system provides parameter files comprising condition symbol definitions. When you write a condition handler, you must specify one of the following parameter files, as appropriate, in an INCLUDE statement:

SYS$LIBRARY:FORDEF.FOR

This file contains definitions for all condition symbols from the FORTRAN-specific Run-Time Library. These symbols have the form:

```
FOR$_xyz
```

For example:

```
FOR$_INPCONERR
```

SYS$LIBRARY:MTHDEF.FOR

This file contains definitions for all condition symbols from the mathematical procedures library. These symbols have the form:

```
MTH$_xyz
```

For example:

```
MTH$_SQUROONEG
```

SYS$LIBRARY:SIGDEF.FOR

This file contains miscellaneous symbol definitions used in FORTRAN condition handlers. These symbols have the form:

```
SS$_xyz
```

For example:

```
SS$_FLTOVF
```

You can obtain a listing of the condition symbols for any facility and their corresponding hexadecimal values by means of the following procedure:

1. Create a file that contains the lines

   ```
   $xxDEF      GLOBAL
   .END
   ```

   To simplify the procedure, make sure the file type is MAR.

   Specify a value for xx according to the set of condition symbols you're concerned with. For example:

   ```
   $SSDEF      GLOBAL
   .END
   ```

2. Enter the command:

   ```
   $ MACRO      file-spec
   ```

   where file-spec specifies the file you created in step 1. If you specified a file type of MAR, you need to enter only the file name with this command.

3. Enter the command:

   ```
   $ LINK/MAP/FULL/NOEXE      file-spec
   ```

   Following execution of this command, you will have a file (file-name MAP) that contains all the symbols defined in xxDEF, in numeric and alphabetic order.

Example:

```
$ EDIT   DEF.MAR
(create file)

$ MACRO   DEF

$ LINK/MAP/FULL/NOEXE   DEF
```

6.3.5 Floating Overflow, Zero Divide Exceptions

Some conditions involving floating point operations require that you take special action if you wish to continue execution. Operations that involve floating overflow, divide by 0, computing the square root of a negative number, etc. store a unique result known as a floating reserved operand. If a subsequent floating point operation accesses this result, a hardware reserved operand fault is generated and signaled. This may continue indefinitely if the reserved operand is not changed and is accessed by each subsequent attempt to perform the computation.

To allow computation to continue, you must change the reserved operand by calling the Run-Time Library routine LIB$FIXUP_FLT, as shown in this example:

```
CALL LIB$FIXUP_FLT(SIGARGS,MECHARGS,1.7D38)
```

The third argument is optional. If you omit it, the reserved operand is changed to +0.0. However, you should include the argument, and specify it as a double precision value. This ensures that the reserved operand will be changed correctly regardless of its precision. For more information on LIB$FIXUP_FLT, see the VAX-11 Common Run-Time Procedure Library Reference Manual.

6.4 CONDITION HANDLER EXAMPLES

The examples in this section illustrate condition handlers that apply to typical FORTRAN procedures.

Example 1:

The following example shows a matrix inversion procedure, using the logical function INVERT to indicate success or failure. That is, if the matrix can be inverted, INVERT returns the logical value .TRUE. If the matrix is singular, INVERT returns the logical value .FALSE. As the matrix inversion procedure is being executed, a floating overflow or divide-by-zero exception may occur. A condition handler (HANDL) is provided to recover from these exceptions and return the value .FALSE. to the calling program. Note that the condition handler is defined as an INTEGER*4 function.

```
      LOGICAL FUNCTION INVERT (A,N)
      DIMENSION A(N,N)
      EXTERNAL HANDL
      CALL LIB$ESTABLISH (HANDL)    ! ESTABLISH HANDLER
      INVERT = .TRUE.               ! ASSUME SUCCESS
              .
              .  (INVERT THE MATRIX)
              .
      RETURN
      END

      INTEGER*4  FUNCTION  HANDL  (SIGARGS, MECHARGS)
      INTEGER*4  SIGARGS(3), MECHARGS(5)
      INCLUDE 'SYS$LIBRARY:SIGDEF'
      HANDL = SS$_RESIGNAL          ! ASSUME RESIGNAL

      IF (SIGARGS(2) .EQ. SS$_FLTOVF .OR.
     1    SIGARGS(2) .EQ. SS$_FLTDIV) THEN
           MECHARGS(4) = .FALSE.
           CALL  SYS$UNWIND(%VAL(0),%VAL(0))
           ENDIF

      RETURN
      END
```

If an exception occurs during the execution of INVERT, the condition handler (HANDL) is called. The handler must first determine whether it can deal with the condition being signaled. Thus, the condition handler tests the condition value (SIGARGS(2)). If the condition is floating overflow or floating division by 0, the condition handler causes a return from INVERT with the value .FALSE.

The condition handler uses the unwind procedure to force a return to the procedure that called INVERT. The logical value .FALSE. is stored in the saved R0 element of the mechanism vector (MECHARGS(4)). This value is used as the function value for INVERT when the unwind occurs. The handler calls SYS$UNWIND and returns; the condition handling facility then gets control and actually performs the unwind operation. Note that the function value from the user condition handler (HANDL = SS$_RESIGNAL) is ignored if SYS$UNWIND is called.

If the exception condition is not a floating overflow or division by 0, the condition handler returns a value of SS$_RESIGNAL, indicating that it is not able to deal directly with the condition. The immediately preceding procedure activation will then be checked for a condition handler; continuing until an established condition handler or the default condition handler is reached.

Example 2:

The following example illustrates a condition handler that processes the Run-Time Library conditions MTH$_FLOOVEMAT and MTH$_FLOUNDMAT. The purpose of the condition handler is to modify the value returned by the Run-Time Library from the default value (reserved operand -0.0) to 0.0, and to suppress the printing of an error message.

```
C  MAIN PROGRAM
   EXTERNAL HDLR
   CALL LIB$ESTABLISH (HDLR)
      .
      .
      .
   X = EXP(Y)
      .
      .
      .
   END

   INTEGER*4  FUNCTION HDLR (SIGARGS,MECHARGS)
   INTEGER*4  SIGARGS(2), MECHARGS(5)
   INCLUDE 'SYS$LIBRARY:SIGDEF'
   INCLUDE 'SYS$LIBRARY:MTHDEF'

   IF (SIGARGS(2) .EQ. MTH$_FLOOVEMAT .OR.
  1    SIGARGS(2) .EQ. MTH$_FLOUNDMAT) THEN

      MECHARGS(4) = 0
      MECHARGS(5) = 0
      HDLR = SS$_CONTINUE
   ELSE
      HDLR = SS$_RESIGNAL
   ENDIF

   RETURN
   END
```

When an exception condition occurs, HDLR is called, and the contents of the second element of the signal vector are compared with MTH$_FLOOVEMAT and MTH$_FLOUNDMAT. If either of the specified conditions is detected, 0's are placed in the saved R0 and R1 elements of array MECHARGS, the value SS$_CONTINUE is returned in HDLR, and execution continues in the math library and no error message is printed. R0 and R1 are returned. If the condition is not one of those specified, SS$_RESIGNAL is returned, and the preceding procedure activations are searched for an established condition handler.

CHAPTER 7

FORTRAN SYSTEM ENVIRONMENT

This chapter discusses aspects of the relationship between VAX-11 FORTRAN IV-PLUS and the VAX-11 system. The purpose is to provide insights that will permit you to use VAX-11 FORTRAN IV-PLUS in a way that makes best use of its features. The following subjects are discussed:

- Program sections
- Storage allocation
- FORTRAN-supplied functions
- DO loops
- ENTRY statement arguments
- Floating point data representation

7.1 PROGRAM SECTION USAGE

The storage required by a VAX-11 FORTRAN IV-PLUS program unit is allocated in contiguous areas called program sections (PSECTs). The VAX-11 FORTRAN IV-PLUS compiler implicitly declares three PSECTs, named:

$CODE - This program section contains all executable code.

$PDATA - This program section contains read-only data (e.g., constants and FORMAT statements).

$LOCAL - This program section contains read/write data local to the program unit.

In addition, each COMMON block you declare causes allocation of a PSECT with the same name as the COMMON block. (The unnamed COMMON block PSECT is named $BLANK.)

The linker controls memory allocation and sharing according to the attributes of each PSECT. See Table 7-1.

Each module comprised by your program is named according to the name specified in the PROGRAM, BLOCK DATA, FUNCTION, or SUBROUTINE statement used in creating the module. You can use the module name to qualify the PSECT name specified in LINK commands. Refer to the VAX-11 Linker Reference Manual.

Defaults are applied to PROGRAM and BLOCK DATA statements. They are:

```
PROGRAM default    - source-file-name$MAIN
BLOCK DATA default - source-file-name$DATA
```

Table 7-1
PSECT Names and Attributes

PSECT Name	Use	Attributes
$CODE	Executable code.	PIC,CON,REL,LCL,SHR,EXE,RD,NOWRT,LONG
$PDATA	Read-only data: literals, read-only FORMAT statements	PIC,CON,REL,LCL,SHR,NOEXE,RD,NOWRT,LONG
$LOCAL	Read/write data local to the program unit: user local symbols, compiler temporary symbols, argument lists, and descriptors	PIC,CON,REL,LCL,NOSHR,NOEXE,RD,WRT,LONG*
$BLANK	Blank COMMON block.	PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG
name(s)	Named COMMON block(s).	PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG

* Program section $LOCAL is aligned on quadword boundaries if it contains any double precision or complex data.

Table 7-2 describes the meanings of the attributes. Refer also to the VAX-11 Linker Reference Manual.

When the VAX/VMS linker constructs an executable image, it divides the executable image into sections. Each image section contains PSECTs that have the same attributes. By arranging image sections according to PSECT attributes, the linker is able to control memory allocation. The linker allows you to allocate memory to your own specification, by means of commands you include in an options file that is input to the linker. The options file is described in the VAX-11 Linker Reference Manual.

7.2 STORAGE ALLOCATION AND FIXED-POINT DATA TYPES

The default storage unit for VAX-11 FORTRAN IV-PLUS is the longword (four bytes). A storage unit is the amount of memory needed to store a real, logical, or integer value. Double precision and complex values are stored in two successive storage units. These relative sizes must be taken into account when you associate two or more variables through an EQUIVALENCE or COMMON statement, or by argument association.

You can, however, declare integer and logical variables as 2-byte values to save space or to be compatible with PDP-11 FORTRAN. Either specify the /NOI4 qualifier in the FORTRAN command, or explicitly declare a variable as INTEGER*2 or LOGICAL*2 type. This allows you to take advantage of VAX-11's ability to manipulate both 16-bit data and 32-bit data efficiently.

Table 7-2
PSECT Attributes

Attribute	Meaning
PIC/NOPIC	Position-independent or position-dependent.
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
SHR/NOSHR	Shareable or non-shareable
EXE/NOEXE	Executable or non-executable
RD/NORD	Readable or non-readable
WRT/NOWRT	Writeable or non-writeable
LONG/QUAD	Longword or quadword alignment

7.2.1 Integer Data Types Supported

VAX-11 FORTRAN IV-PLUS supports INTEGER*2 and INTEGER*4 data types, which occupy two and four bytes of storage respectively. Both types can be mixed in computations; such mixed type computations are carried out to 32 bits of significance, and produce INTEGER*4 results.

If you do not override the default storage allocation, then four bytes are allocated for integer values.

7.2.1.1 **Relationship of INTEGER*2 and INTEGER*4 Values** - INTEGER*2 values are stored as two's complement signed binary numbers, and occupy two bytes of storage. INTEGER*4 values are also stored as two's complement signed binary numbers, but occupy four bytes of storage. The lower addressed word of an INTEGER*4 value contains the low-order part of the value.

INTEGER*2 values are a subset of INTEGER*4 values. That is, an INTEGER*4 value in the range -32768 to 32767 can be treated as an INTEGER*2 value. Conversion from INTEGER*4 to INTEGER*2 (without checks for overflow) consists of simply ignoring the high-order 16 bits of the INTEGER*4 value. This type of conversion provides an important FORTRAN usage, as illustrated in the following example. Given:

```
CALL SUB(2)
```

should the argument (2) be treated as an INTEGER*2 value or as INTEGER*4? By providing an INTEGER*4 constant as the actual argument, SUB executes correctly even if its dummy argument is typed INTEGER*2.

7.2.1.2 **Integer Constant Typing** - Integer constants are generally typed according to the magnitude of the constant. In most contexts, INTEGER*2 and INTEGER*4 variables and integer constants can be freely mixed. You are responsible, however, for ensuring that integer overflow conditions do not occur, as happens in the following example.

```
INTEGER*2 I
INTEGER*4 J
I = 32767
J = I + 3
```

In this example, I and 3 are INTEGER*2 values, and an INTEGER*2 result will be computed. The 16-bit addition, however, will overflow the valid INTEGER*2 range and be treated as -32766. This value will be converted to INTEGER*4 type and assigned to J. The overflow will be detected and reported if the program unit was compiled with the default /CHECK=OVERFLOW qualifier specified.

Contrast the preceding example with the following apparently equivalent program, which produces different results.

```
PARAMETER I = 32767
INTEGER*4 J
J = I + 3
```

In this case the compiler performs the addition of the constant 3 and the parameter constant 32767, producing a constant result of 32770. The compiler recognizes this as an INTEGER*4 value. Thus, J will be assigned the value 32770.

7.2.1.3 **Integer-Valued Processor-Defined Functions** - A number of the processor-defined functions provided by FORTRAN (see Section 7.3) produce integer results from real or double precision arguments; for example, INT. In order to support such functions in a manner compatible with both INTEGER*2 and INTEGER*4 modes, two versions of these integer-valued processor-defined functions are supplied. The compiler chooses the version that matches the compiler /I4 qualifier setting (/I4 or /NOI4). This is similar to generic function selection (described in Section 7.3.1), except that the selection is based on the mode of the compiler, rather than on the argument data type.

In some cases, you may need to use the version of an integer-valued processor-defined function that is the opposite of the compiler qualifier setting. For this reason, a pair of additional processor-defined function names are provided for each standard integer-valued processor-defined function. The names of the INTEGER*2 versions are prefixed with I, and the names of the INTEGER*4 versions with J (for example, IIABS and JIABS). See Appendix B of the VAX-11 FORTRAN IV-PLUS Language Reference Manual for a complete list of processor-defined functions.

7.2.2 Byte (LOGICAL*1) Data Type

The FORTRAN IV-PLUS byte data type lets you take advantage of the byte processing capabilities of VAX-11. Actually, BYTE or LOGICAL*1 is a signed integer data type, and is useful for storing and manipulating Hollerith data.

In general, when data of different types are used in a binary operation, the lower-ranked type is converted to the higher-ranked

type prior to computation. (Data type rank is discussed in the VAX-11 FORTRAN IV-PLUS Language Reference Manual.) However, in the case of a byte variable and an integer constant in the range representable as a byte variable (-128 to 127), the integer constant is treated as a byte constant; and the result is also of byte data type.

7.3 FUNCTIONS SUPPLIED WITH VAX-11 FORTRAN

VAX-11 FORTRAN IV-PLUS includes a number of processor-defined functions. These are listed in Appendix B of the VAX-11 FORTRAN IV-PLUS Language Reference Manual. For descriptions of the algorithms used to generate processor-defined functions, refer to the VAX-11 Common Run-Time Procedure Library Reference Manual.

Processor-defined functions include routines provided by VAX-11 FORTRAN IV-PLUS to perform commonly-used mathematical functions (such as COS and EXP) and utility services (such as DATE and TIME). The data type of a processor-defined function is unaffected by the use of the IMPLICIT statement to change the default data type. Some of these processor-defined functions are generic functions; that is, you can refer to a set of similar routines by one common name.

7.3.1 Generic Functions

Many of the functions supplied with VAX-11 FORTRAN IV-PLUS are generic functions, which means that you refer to them by a common name, to perform much the same function. For example, there are three processor-defined functions that calculate cosines, all of which are (or can be) referred to by the generic name COS; their names are COS, DCOS, and CCOS. They differ in that they return single precision, double precision, or complex values, respectively. If you request that the cosine function be invoked, you need only refer to it generically, as COS. The compiler selects the appropriate routine based on the arguments you specify. If the argument is single precision, COS is selected; if it is double precision, DCOS is selected; and if complex, CCOS is selected.

Note, however, that you can explicitly refer to a particular routine if you wish. Thus, to obtain the double precision cosine function, you could specify DCOS, rather than using the generic name.

The compiler provides the names of processor-defined functions it selects by listing them by their internal names in the "FUNCTIONS AND SUBROUTINES REFERENCED" section of the listing.

7.3.2 Use of the EXTERNAL Statement

The EXTERNAL statement has special applicability in conjunction with processor-defined functions. If you refer to a subprogram by means of one of the processor-defined function names, the compiler assumes that you intend that the processor-defined function be used. Thus, if one of your own subprograms has the same name as a system-supplied function or subroutine, you must distinguish your routine's name from the processor-defined function name.

To do so, specify the name in an EXTERNAL statement as follows:

```
EXTERNAL *name
```

The compiler interprets any name listed in an EXTERNAL statement, that is prefixed by an asterisk, as the name of a user-supplied subprogram.

The EXTERNAL statement is described in the VAX-11 FORTRAN IV-PLUS Language Reference Manual.

7.4 ITERATION COUNT MODEL FOR DO LOOPS

VAX-11 FORTRAN IV-PLUS provides an extended form of the DO statement, with the following features:

- The control variable can be an integer, real, or double precision variable.
- The initial value, step size, and final value of the control variable can be any expression that produces an integer, real, or double precision result.
- The number of times the loop is executed (the iteration count) is determined at the initialization of the DO statement, and is not re-evaluated during successive executions of the loop. Thus, the number of times the loop is executed is not affected by changing the values of the parameter variables used in the DO statement.

7.4.1 Cautions Concerning Program Interchange

Some common practices associated with the use of DO statements on other FORTRAN systems may not have the intended effects when used with VAX-11 FORTRAN IV-PLUS. For example:

- Assigning a value to the control variable within the body of the loop that is greater than the final value, in order to cause early termination of the loop.
- Similarly modifying either a step size variable or a final value variable to either modify the loop behavior or terminate the loop.
- Using a negative or zero step size to cause an arbitrarily long loop that is terminated by a conditional control transfer within the loop.

7.4.2 Iteration Count Computation

Given the following sample DO statement

```
DO label, V=m1,m2,m3
```

(where m1, m2, and m3 are any expressions), the iteration count is computed as follows:

```
count= MAX(1,INT(((m2-m1)/m3)+1))
```

This computation:

- Provides that the body of the DO loop is always executed at least once

- Permits the step size (m3) to be negative or positive, but not zero

- Gives a well-defined and predictable count value for expressions resulting from any combination of the allowed result types. Note, however, that the effects of round-off error inherent in any floating point computation may cause the count to be greater or less than desired when real or double precision values are used.

Under certain conditions it is not necessary to compute the iteration count explicitly. For example, if all of the parameters are of type integer and if the parameter values are not modified in the loop, then the FORTRAN IV-PLUS generated code controls the number of iterations of the loop by comparing the control variable directly with the final value.

7.5 ENTRY STATEMENT ARGUMENTS

The association of actual and dummy arguments is described in Chapter 6 of the VAX-11 FORTRAN IV-PLUS Language Reference Manual. In general, that description suffices for most cases. However, the implementation of argument association in ENTRY statements varies from the way this is done on some other FORTRAN systems.

As described previously in this manual (Chapter 5), VAX-11 FORTRAN uses the call-by-reference and call-by-descriptor methods to pass arguments to called procedures (for numeric and character arguments, respectively). Some other FORTRAN implementations use the call-by-value/result method. This distinction becomes crucial when reference is made to dummy arguments in ENTRY statements.

While standard FORTRAN allows you to use the same dummy arguments in different ENTRY statements, it permits you to refer only to those dummy arguments that are defined for the ENTRY point being called. For example:

```
SUBROUTINE SUB1(X,Y,Z)
   .
   .
   .
ENTRY ENT1(X,A)
   .
   .
   .
ENTRY ENT2(B,Z,Y)
```

Given this, you can make the following references:

CALL	Valid References
SUB1	X Y Z
ENT1	X A
ENT2	B Z Y

FORTRAN implementations that use the call-by-value/result method, however, permit you to refer to dummy arguments that are not defined in the ENTRY statement being called. For example:

```
SUBROUTINE INIT(A,B,C)
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

You can use this non-standard device in call-by-value/result implementations, because a separate internal variable is allocated for each dummy argument in the called procedure. When the procedure is called, each scalar actual argument value is assigned to the corresponding internal variable, and these variables are then used whenever there is a reference to a dummy argument within the procedure. On return from the called procedure, modified dummy arguments are copied back to the corresponding actual argument variables.

When an entry point is referenced, all its dummy arguments are defined with the values of the corresponding actual arguments, and may be referenced on subsequent calls to the subprogram. However, you are advised not to attempt to do this in programs that are to be executed on VAX-11 FORTRAN. Such references will not have the intended effect on VAX-11, and will produce programs that are not transportable to other systems that use the call-by-reference (descriptor) method.

VAX-11 creates associations between dummy and actual arguments by passing the address of each actual argument, or descriptor, to the called procedure. Each reference to a dummy argument generates an indirect address reference through the actual argument address. When control returns from the called procedure, the association between actual and dummy arguments ends. The dummy arguments do not retain their values, and therefore cannot be referenced on subsequent calls. Thus, to perform the sort of non-standard references shown in the previous example, the subprogram must copy the values of the dummy arguments. For example:

```
SUBROUTINE INIT(A1,B1,C1)
A = A1
B = B1
C = C1
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

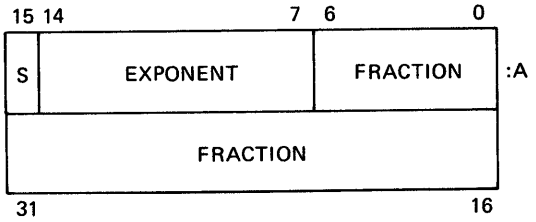
This will work on VAX-11, and on systems that use the call-by-value/result method. However, this method should also be avoided, because it is also non-standard. The success of this example depends on the storage for A, B, and C being statically allocated so they will retain their values from one call to the next.

7.6 FLOATING POINT DATA REPRESENTATION

The following sections describe how floating point data is represented internally.

7.6.1 Single Precision Floating Point Data

A single precision floating point value is represented by four contiguous bytes. The bits are numbered from the right 0 through 31.



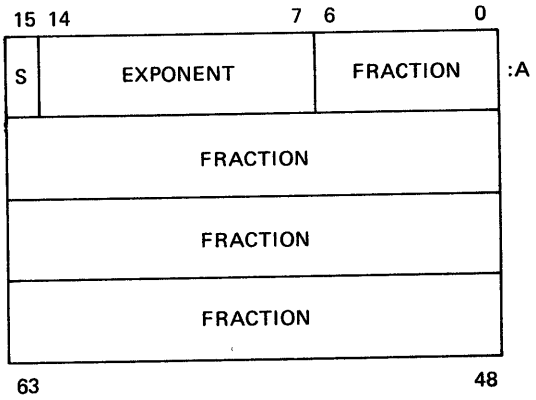
A single precision floating point value is specified by its address A, the address of the byte containing bit 0. The form of the value is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and 0 through 6.

The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0, indicates that the floating point value has a value of 0. Exponent values of 1 through 255 indicate binary exponents of -127 through +127. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating point instructions processing a reserved operand take a reserved operand fault (see Section 7.6.3.1).

The value of a floating point value is in the approximate range of .29*10**-38 through 1.7*10**38. The precision of a floating point value is approximately one part in 2**23, i.e., typically 7 decimal digits.

7.6.2 Double Precision Floating Point Data

A double precision floating point value is represented by eight contiguous bytes. The bits are numbered from the right 0 through 63.



A double precision floating point value is specified by its address A, the address of the byte containing bit 0. The form of a double precision floating point value is identical to a single precision floating point value except for an additional 32 low significance fraction bits. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 6.

The exponent conventions and approximate range of values are the same for double precision floating point values as single precision floating point values. The precision of a double precision floating point value is approximately one part in 2**55, i.e., typically 16 decimal digits.

7.6.3 Floating Point Data Characteristics

Certain FORTRAN programming practices that are commonly used, though not permitted under the rules for standard FORTRAN, may not produce the expected behavior when attempted in VAX-11 FORTRAN. These are described below.

7.6.3.1 **Reserved Operand Faults** - Floating point variables that contain invalid floating point values (-0.0), indicated by an exponent field of 0 and a sign bit of 1, cause a reserved operand fault in the VAX-11 hardware. An error is reported, and, by default, your program terminates.

There are three ways to create reserved operand values:

- The VAX-11 hardware stores a reserved operand value as the result of the floating point arithmetic exceptions, floating overflow and floating zero divide.

- The mathematical function library returns a reserved operand value if the function is called incorrectly or if the argument is invalid. For example,

```
SQRT(-1.0)
```

  These return values can be modified by providing a condition handler (see Chapter 6).

- Integer arithmetic and logical operations can create reserved operand bit patterns in floating point variables and arrays associated with integers. Associations of this kind can occur through EQUIVALENCE, COMMON, or argument association. For example:

```
EQUIVALENCE (X,I)
I = 32768
X = X+1.0
```

  Adding 1.0 to X will cause a reserved operand fault, because the integer value 32768 is a reserved operand when interpreted as a floating point value.

The first two cases occur when invalid programs or data are used. The last case can occur inadvertently in a program, and not be detected by other FORTRAN systems.

7.6.3.2 **Representation of 0.0** - The VAX-11 hardware defines 0.0 as any bit pattern that has an exponent field and sign bit of 0, regardless of the value of the fraction. When a bit pattern that is defined as 0.0 is used in a floating point operation, the VAX-11 hardware sets the fraction field to 0. One possible effect is that non-zero integers that are equivalenced to floating point values may be interpreted as 0.

Logical operations can have a similar effect, as shown in the following example:

```
EQUIVALENCE (X,I)
I = 64
IF (X .EQ.  0) GO TO 10
```

The branch will always be taken because the bit pattern that represents the integer value is equivalent to 0 when interpreted as a floating point value.

7.6.3.3 **Sign Bit Tests** - The bit used as the sign bit of a floating point value is not the same bit as the sign bit of an equivalenced INTEGER*4 value. Consequently, you must test the sign of a value by testing the correct data type. For example:

```
EQUIVALENCE (X,I)
I = 40000
IF (X .GT.  0) GO TO 10
```

The branch will not be taken, because the bit pattern that represents the integer value 40000 is negative (bit 15 is set) when interpreted as a floating point value.

CHAPTER 8

PROGRAMMING CONSIDERATIONS

This chapter discusses methods you can use to write efficient source programs. Topics covered are:

- Creating efficient source programs
- Compiler optimization
- I/O system characteristics

8.1 CREATING EFFICIENT SOURCE PROGRAMS

You can reduce the time and memory required for your source programs by taking advantage of features included in the FORTRAN IV-PLUS language, as described below.

8.1.1 PARAMETER Statement

The PARAMETER statement allows you to assign a symbolic name to a constant. For example:

```
PARAMETER PI=3.1415927
```

You can then use the symbolic name (PI) to represent the constant (3.1415927) anywhere a constant is valid (such as in expressions).

Using the PARAMETER statement promotes more efficient object code by allowing constants to be used instead of variables, while permitting easy program modification. Constants can generally be compiled into more efficient code. (See Section 8.2.2.) Therefore, parametric variables should be defined by means of PARAMETER statements, rather than by means of DATA or assignment statements. The source code illustrated in Example A will produce more efficient code than either Example B or Example C.

Example A

```
      PARAMETER M = 50, N = 100
      DIMENSION X(M), Y(N)
      DO 5, I = 1, M
      DO 5, J = 1, N
5     X(I) = X(I)*Y(J) + X(M)*Y(N)
```

Example B

```
      DIMENSION X(50), Y(100)
      DATA M, N/50,100/
      DO 5, I = 1, M
      DO 5, J = 1, N
5     X(I) = X(I)*Y(J) + X(M)*Y(N)
```

Example C

```
      DIMENSION X(50), Y(100)
      M = 50
      N = 100
      DO 5, I = 1, M
      DO 5, J = 1, N
5     X(I) = X(I)*Y(J) + X(M)*Y(N)
```

8.1.2 INCLUDE Statement

The INCLUDE statement is used to incorporate a file into your source program. This allows you to avoid duplicating redundant code in source programs. For example, there may be several lines of source code, such as a COMMON block specification, that appear in several program units. Rather than repeat this code in each program unit, you can create a separate file that consists of the code; then specify an INCLUDE statement in each program unit that requires the code.

Example

In this example, a COMMON block specification is required in a program as well as in a subroutine called by the program. The COMMON block specification is put into a file (COMMON.FOR), and an INCLUDE statement is used in both the program and the subroutine, to reference the code. The file, COMMON.FOR, consists of the following text:

```
      PARAMETER M = 100
      COMMON X(M), Y(M)
```

Main Program

```
      INCLUDE 'COMMON.FOR'
      DIMENSION Z(M)
      CALL CUBE
      DO 5 I = 1,M
5     Z(I) = X(I) + SQRT(Y(I))
            .
            .
            .
      SUBROUTINE CUBE
      INCLUDE 'COMMON.FOR'
      DO 10 I = 1,M
10    X(I) = Y(I)**3
      RETURN
      END
```

The file COMMON.FOR defines the size of the COMMON block and the sizes of the arrays X, Y, and Z. Any changes to the COMMON block will be reflected automatically after recompilation.

8.1.3 Allocating Variables in Common Blocks

When you allocate variables in a common block, you should do so in such a way that they are aligned on natural boundaries in memory. One simple method to accomplish this is to allocate variables in order, according to data type. First allocate INTEGER*4, LOGICAL*4, REAL, and DOUBLE PRECISION variables; then INTEGER*2, LOGICAL*2 variables; and finally BYTE and CHARACTER variables.

8.1.4 Conditional Branching

You will generally produce more efficient programs if you use IF...THEN...ELSE statements to control program flow than if you use IF... GO TO statements.

8.2 COMPILER OPTIMIZATIONS

Optimization refers to techniques used to produce the greatest amount of processing with the least amount of time and memory. The aim is to create programs that are efficient in speed and size. Optimum efficiency results from carefully designed and written programs, and from compilation techniques that take advantage of the machine architecture. You can produce optimum source programs by being aware of, and using, certain features provided by the VAX-11 FORTRAN IV-PLUS language; the VAX-11 FORTRAN IV-PLUS compiler produces efficient code by deriving maximum benefit from the VAX-11 hardware.

The primary goal of VAX-11 FORTRAN IV-PLUS optimizations is to produce an object program that executes faster than an unoptimized version of the same source program. A secondary goal is to reduce the size of the object program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization. The VAX-11 FORTRAN IV-PLUS compiler performs the following optimizations:

- Evaluation at compile time of integer, real, and double precision constant expressions involving addition, subtraction, multiplication, or division (this technique is called "constant folding").
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array element references.
- Argument-list merging. If two subprogram references have the same arguments, a single copy of the argument list is generated.
- Branch instruction optimizations for arithmetic, logical and block IF statements.
- Elimination of unreachable code. An optional warning message is issued to indicate unreachable statements in the source program.

- Recognition and replacement of common subexpressions.
- Removal of invariant computations from DO loops.
- Local register assignment. Frequently-referenced variables are retained (if possible) in registers to reduce the number of memory references needed.
- Assignment of frequently-used variables and expressions to registers across DO loops.
- Assignment of base registers, to provide shorter address references to COMMON blocks.
- Constant pooling. Storage is allocated for only one copy of a constant in the compiled program. Constants used as immediate-mode operands are not allocated storage. This includes most numeric constants.
- In-line code expansion for some processor-defined functions.
- Fast calling sequences for the real and double precision versions of some processor-defined functions.
- Reordering the evaluation of expressions, to minimize the number of temporary values required.
- Delaying unary minus and .NOT. operations to eliminate unary negation/complement operations.
- Partial evaluation of Boolean expressions. For example, if el in the following expression has the value .FALSE., e2 is not evaluated:

```
IF(el.AND.e2) GO TO 20
```

- Optimization of control transfers.
- Peephole optimization of instruction sequences; i.e., examining code on an instruction-by-instruction basis to find operations that can be replaced by shorter, faster equivalent operations.

8.2.1 Characteristics of Optimized Programs

Optimized programs produce results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different source program statements. For example:

Unoptimized Program

```
      A = X/Y
      B = X/Y
      DO 10, I = 1,10
10    C(I) = C(I)*(X/Y)
```

Optimized Program

```
      t = X/Y
      A = t
      B = t
      DO 10, I = 1,10
10    C(I) = C(I)*t
```

In the example above, if Y has the value 0.0, the unoptimized program produces 12 zero-divide errors at run time; the optimized program produces 1. (Note that t is a temporary variable created by the compiler.) Eliminating redundant calculations and moving invariant calculations out of loops can affect detection of such arithmetic errors, and should be kept in mind when you include error-detecting routines in your program.

8.2.2 Compile-Time Operations on Constants

The compiler performs the following computations on expressions involving constants (including PARAMETER constants) at compile time.

- Negation of Constants: Constants preceded by unary minus signs are negated at compile time.

  For example:

  ```
  X = -10.0
  ```

  is compiled as a single move instruction.

- Type Conversion of Constants: Lower ranked constants are converted to the data type of the higher ranked operand at compile time.

  For example:

  ```
  X = 10*Y
  ```

  is compiled as

  ```
  X = 10.0*Y
  ```

- Arithmetic on Integer, Real, and Double Precision Constants: Expressions involving +, -, *, or / operators are evaluated at compile time.

  For example:

  ```
  PARAMETER NN = 27
  I = 2*NN+J
  ```

  is compiled as

  ```
  I = 54+J
  ```

Array subscript calculations involving constants are simplified at compile time wherever possible.

For example:

```
DIMENSION I(10,10)
I(1,2) = I(4,5)
```

is compiled as a single move instruction.

8.2.3 Source Program Blocks

Some optimizations are performed within the confines of a single block of the source program, where a block consists of a sequence of one or more FORTRAN source statements. The start of a block is generally defined by a labeled statement that is the target of a control transfer from another statement (GO TO, arithmetic IF, ERR=option); or by an ENTRY statement. The following kinds of statement labels, however, do NOT generally define the start of a new block:

- unreferenced statement labels
- a label terminating a DO loop, if the only references to the label occur in DO statements
- labels of FORMAT statements; FORMAT statements must be labeled, but control cannot be transferred to a FORMAT statement
- labels with a single reference that precedes the label. For example:

```
      IF (I.EQ.0) GO TO 40
       .
       .
       .
40 CONTINUE
```

- labels for which the only reference is in an immediately preceding arithmetic or logical IF statement. For example:

```
      IF (A) 10,20,20
10    X = 1
```

A block can contain one or more DO loops, as long as none of the labels within the loops defines the start of a new block. Thus the following examples are considered single blocks and are optimized as complete units:

Example 1

```
      X = B*C
      DO 10, I=1,N
10    A(I) = A(I)/(B*C)
      DO 20, J=1,N
20    Y(J) = Y(J)+B*C
```

Example 2

```
      DO 20, I=1,N
      DO 20, J=1,N
      SUM = 0.0
      DO 10, K=1,N
10    SUM = SUM+A(I,K)*B(K,J)
20    C(I,J) = SUM
```

A more thoroughly optimized object program is produced if the number of separate blocks is minimized. Common subexpression, code motion, and register allocation optimizations are performed within single blocks.

8.2.4 Eliminating Common Subexpressions

The same subexpression often appears in more than one computation within a program. For example:

```
A = B*C+E*F
      .
      .
      .
H = A+G-B*C
      .
      .
      .
IF ((B*C)-H)10,20,30
```

In this code sequence, the common subexpression B*C appears three times. If the values of the operands of this subexpression do not change between computations, its value can be computed once and the result can be used in place of the subexpression. Thus, the sequence shown above is compiled as follows:

```
t = B*C
A = t+E*F
      .
      .
      .
H = A+G-t
      .
      .
      .
IF ((t)-H)10,20,30
```

As you can see, two computations of B*C have been eliminated.

Of course, you could have optimized the source program itself to preclude the redundant calculation of B*C. The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect. Without optimization, the statements

```
DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)
```

can be compiled as

```
t1 = J*25+I
t2 = J*25+I
A(t1)=B(t2)
```

Variables t1 and t2 represent equivalent expressions. The compiler eliminates this redundancy by producing the following optimization:

```
t = J*25+I
A(t)=B(t)
```

8.2.5 Removing Invariant Computations from Loops

Execution speed is enhanced by taking invariant computations out of loops. For example, if the compiler detected the following sequence

```
      DO 10, I = 1,100
10    F = 3.0*Q*A(I)+F
```

it would recognize that the subexpression 3.0*Q has the same value each time the loop is executed. Thus, it would change the sequence to

```
      t = 3.0*Q
      DO 10, I = 1,100
10    F = t*A(I)+F
```

This moves the calculation of 3.0*Q out of the loop, and saves 99 multiply operations.

8.2.6 Compiler Optimization Example

Figure 8-1 illustrates many of the optimization techniques used by the VAX-11 FORTRAN IV-PLUS compiler. The first part (Figure 8-1a) shows a complete VAX-11 FORTRAN subroutine, a relaxation function often used in engineering applications. This subroutine is a 2-dimensional function used to obtain the values of a variable at coordinates on a surface; for example, temperatures distributed across a metal plate.

The second part (Figure 8-1b) shows the VAX-11 machine code generated by the FORTRAN compiler. Several compiler optimizations are illustrated, as noted by circled numbers next to the generated code lines. These are described in the notes that follow the figure.

```
0001                SUBROUTINE RELAX2(EPS)

0002                PARAMETER M=40, N=60
0003                DIMENSION X(0:M,0:N)
0004                COMMON X

0005                LOGICAL DONE

0006      1         DONE = .TRUE.

0007                DO 10 J = 1,N-1
0008                DO 10 I = 1,M-1
0009                    XNEW = ( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) ) / 4
0010                    IF ( ABS(XNEW-X(I,J)) .GT. EPS ) DONE = .FALSE.
0011      10            X(I,J) = XNEW

0012                IF (.NOT. DONE) GO TO 1

0013                RETURN
0014                END
```

Figure 8-1a RELAX Source Program

```
                .TITLE  RELAX2
                .IDENT  01

0000            .PSECT  $BLANK
0000    X:

0000            .PSECT  $CODE
0000    RELAX2::
0000            .WORD   ^M<IV,R5,R6,R7,R8,R9,R10,R11>
0002            MOVAL   $LOCAL, R11
                                                                ; 0006
0009    .1:
0009            MNEGL   #1, DONE(R11)
                                                                ; 0007
000C            MOVL    #1, R7          (1)
000F            MOVAL   $BLANK, R5      (2)
0016    L$IANE:
                                                                ; 0008
0016            MOVL    #1, R9          (3)
0019            MULL3   #41, R7, R6     (4)
001D    L$IAGG:
                                                                ; 0009
001D            ADDL3   R9, R6, R10     (5)
0021            ADDF3   X+4(R5)[R10], X-4(R5)[R10], R0
0029            ADDF2   X-164(R5)[R10], R0
002F            ADDF2   X+164(R5)[R10], R0
0035            MULF3   #^X3F80, R0, R8 (6)
                                                                ; 0010
003D            SUBF3   X(R5)[R10], R8, R0
0042            BICW2   #^X8000, R0     (7)
0047            CMPF    R0, @EPS(AP)
004B            BLEQ    L$IAPI
004D            CLRL    DONE(R11)
004F    L$IAPI:
                                                                ; 0011
004F            MOVL    R8, X(R5)[R10]
0053            AOBLEQ  #39, R9, L$IAGG (8)
0057            AOBLEQ  #59, R7, L$IANE
005B            MOVL    R7, J(R11)
005F            MOVL    R8, XNEW(R11)
0063            MOVL    R9, I(R11)
                                                                ; 0012
0067            BLBC    DONE(R11), .1
006A            RET
                .END
```

Figure 8-1b RELAX Machine Code (Optimized)

Notes for Figure 8-1

(1) Register assignment for J

(2) Register assignment for a base register for blank COMMON

(3) Register assignment for I

(4) Invariant computation (J*41) removed from the inner loop and assigned to a register

(5) Common subexpression evaluation and local register assignment; 6 uses are made of the value

Notes for Figure 8-1 (Cont.)

6. Peephole optimization; a divide by 4.0 is replaced by a multiply by 0.25.

7. Inline ABS function

8. DO loop control using the Add One and Branch Less Than or Equal (AOBLEQ) instruction

8.3 FORTRAN I/O SYSTEM CHARACTERISTICS

You can often reduce the execution time of your FORTRAN programs by making use of the following facts relevant to the FORTRAN I/O subsystem.

- Unformatted I/O is substantially faster and more accurate than formatted I/O. The unformatted data representation usually occupies less file storage space as well. Thus unformatted I/O should be used for storing intermediate results on secondary storage.

- Specifying an array name in an I/O list is more efficient than using an equivalent implied DO list. A single I/O transmission call passes an entire array, while an implied DO list can pass only a single array element per I/O call.

- The implementation of the BACKSPACE statement involves repositioning the file and scanning previously processed records. If a reread capability is required, it is more efficient to read the record into a temporary array and DECODE the array several times than to read and backspace the record.

- To obtain minimum I/O processing, the record length of direct access sequential organization files should be a divisor or multiple of the device block size of 512 bytes (e.g., 32 bytes, 64 bytes, etc.). For relative organization files, RMS adds one overhead byte for fixed length records and three overhead bytes for variable length records.

- If the approximate size of the file is known, it is more efficient to allocate disk space when a file is opened than to incrementally extend the file as records are written.

- The use of run-time formats should be minimized. The compiler preprocesses FORMAT statements into an efficient internal form. Run-time formats must be converted into this internal form at run time. In many cases, using variable format expressions will allow the format to vary at run time.

- The BLOCKSIZE keyword can be used in an OPEN statement to obtain large amounts of data with each physical I/O operation, thereby greatly improving the processing of sequentially accessed files. For example, specifying a BLOCKSIZE of 4096 bytes results in the transfer of 8 disk blocks for each I/O operation.

The OPEN and CLOSE statements provide explicit control over I/O devices and files. Using these statements in the proper manner can help create efficient source programs, as illustrated in the following examples.

- OPEN (UNIT=1, TYPE='NEW', INITIALSIZE=200)

  This statement allocates space for a file when the file is opened, which is more efficient than extending the size of the file dynamically.

- OPEN (UNIT=3, TYPE='NEW', BLOCKSIZE=8192)

  This statement specifies a large blocking factor for I/O transfers. If the file is on magnetic tape, the physical tape blocks are 8192 bytes long; if the file is on disk, 16 disk blocks are transferred by each I/O operation, thus enhancing I/O performance, though requiring more memory.

- OPEN (UNIT=J, TYPE='NEW', ...)

```
IF (IERR) CLOSE(UNIT=J, DISP='DELETE')

          .
          .
          .

CLOSE (UNIT=J, DISP='SAVE')
```

  A file is created. However, if an error occurs that makes the file invalid or useless, it is deleted.

- OPEN (UNIT=2, FORM='FORMATTED', CARRIAGECONTROL='LIST')

  This statement creates a file with implicit carriage control. The first character of each record is NOT used for carriage control, thus it can contain actual data.

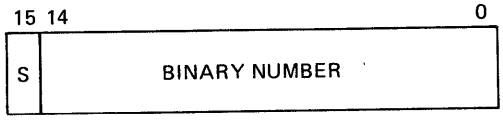
-
```
          CHARACTER*64 FILNAM
  1       TYPE 100
100       FORMAT('$INPUT FILE?')
          ACCEPT 101,FILNAM
101       FORMAT (A)
          OPEN (UNIT=3, NAME=FILNAM, TYPE='OLD', ERR=9)
            .
            .
            .
  9       TYPE 102, FILNAM
102       FORMAT (' ERROR OPENING FILE ',A)
          GO TO 1
```

  This program reads a file specification into the character variable FILNAM. The file is then opened for processing. This permits you to specify the input file name at run time.

APPENDIX A

FORTRAN DATA REPRESENTATION

A.1 INTEGER*2 FORMAT

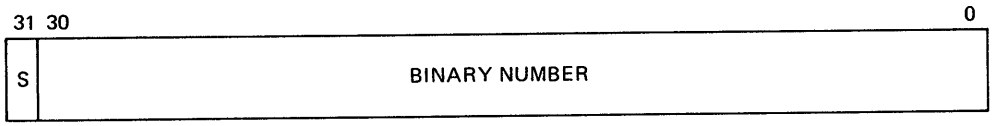


S(sign) = 0(+), 1(-)

Integers are stored in a two's complement representation. INTEGER*2 values lie in the range -32768 to +32767, and are stored in two contiguous bytes aligned on an arbitrary byte boundary. For example:

+22 = 0016(hex)
 -7 = FFF9(hex)

A.2 INTEGER*4 FORMAT



S(sign)=0(+), 1(-)

INTEGER*4 values are stored in two's complement representation. INTEGER*4 values lie in the range -2147483648 to 2147483647. The value is stored in four contiguous bytes, aligned on an arbitrary byte boundary. Note that if the value is in the range of an INTEGER*2 value; i.e., -32768 to +32767, then the first word can be referenced as an INTEGER*2 value.

A.3 FLOATING-POINT FORMATS

The exponent for both floating-point formats is stored in excess 128 notation. Binary exponents from -128 to +127 are represented by the binary equivalents of 0 through 255. Fractions are represented in sign-magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored, because it is redundant (this is called "hidden bit normalization").

This bit is assumed to be a 1 unless the exponent is 0 (corresponding to 2**-128) in which case it is assumed to be 0. The value 0.0 is represented by an exponent field of 0 and a sign bit of 0. For example, +1.0 would be represented in hexadecimal by:

```
4080
0
```

in the 4-byte format, or:

```
4080
0
0
0
```

in the 8-byte format. The decimal number -5.0 is:

```
C0A0
0
```

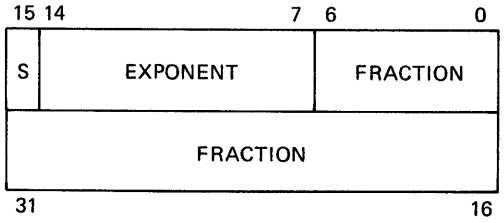
in the 4-byte format, or:

```
C0A0
0
0
0
```

in the 8-byte format.

A.3.1 Real Format (4-Byte Floating Point)

A single precision real number is four contiguous bytes starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 31.

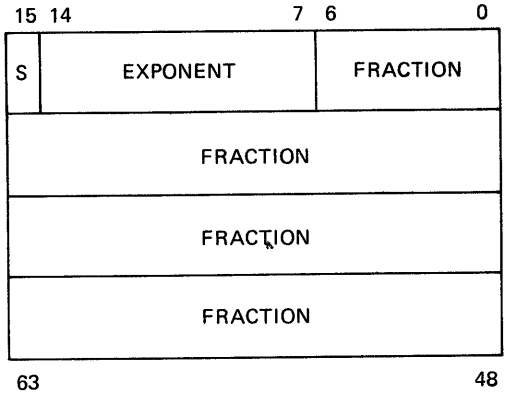


S(sign)=0(+), 1(-)

The form of a single precision real number is sign magnitude, with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. The value of a single precision real number is in the approximate range: .29*10**-38 through 1.7*10**38. The precision is approximately one part in 2**23, i.e., typically 7 decimal digits.

A.3.2 Double Precision Format (8-Byte Floating Point)

A double precision real number is eight contiguous bytes, starting on an arbitrary byte boundary. Bits are labeled from the right, 0 through 63.

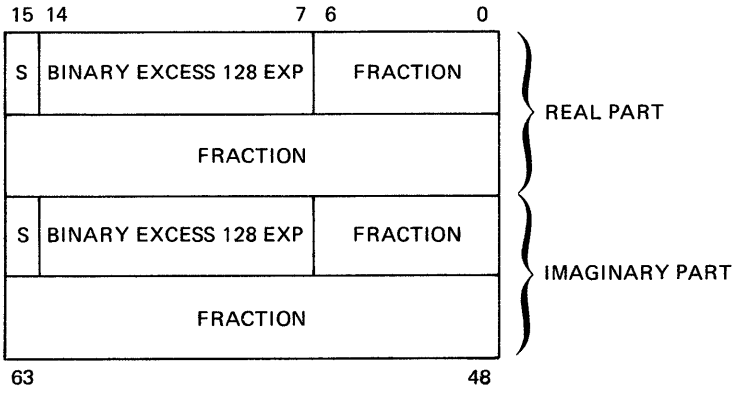


S(sign)=0(+), 1(-)

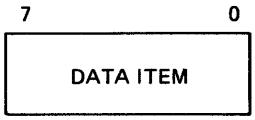
The form of a double precision real number is identical to a single precision real number except for an additional 32 low significance fraction bits. The exponent conventions, and approximate range of values are the same as single precision. The precision is approximately one part in 2**55, i.e., typically 16 decimal digits.

A.3.3 Complex Format

A complex number is eight contiguous bytes, aligned on an arbitrary byte boundary. The low-order four bytes contain a single precision real number that represents the real part of the complex number. The high-order four bytes contain a single precision real number that represents the imaginary part of the complex number.



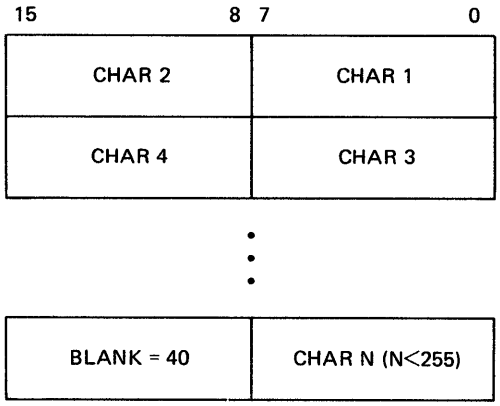
A.4 LOGICAL*1 FORMAT



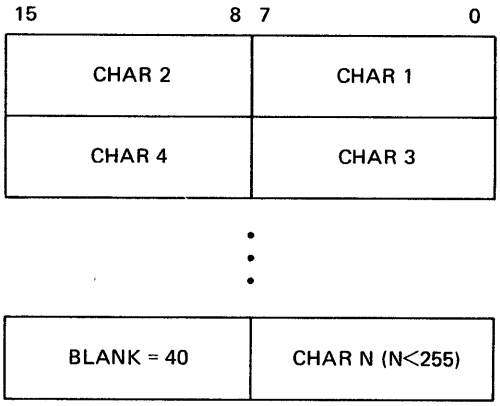
The range of numbers from +127 to -128 can be represented in LOGICAL*1 format.

A.5 CHARACTER FORMAT

Character data is stored as one character per byte, padded with blanks if necessary, to fill the data item.



A.6 HOLLERITH FORMAT

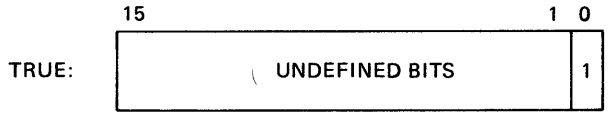


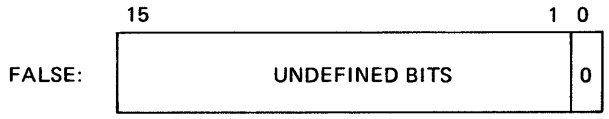
Hollerith constants are stored internally one character per byte. Hollerith values are padded on the right with blanks to fill the associated data item if necessary.

A.7 LOGICAL FORMAT

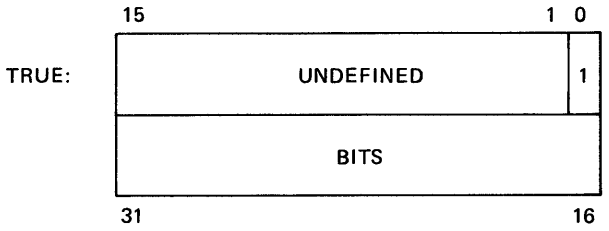
Logical values are stored in two or four contiguous bytes, starting on an arbitary byte boundary. The low-order bit (bit 0) determines the value. If bit 0 is set, the value is .TRUE.. If bit 0 is clear, the value is .FALSE..

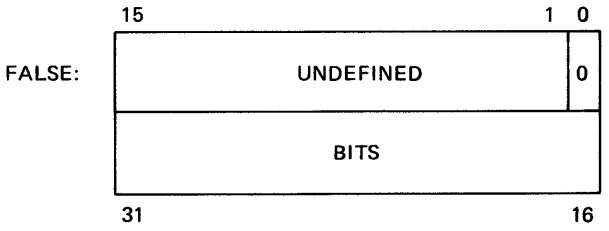
LOGICAL*2





LOGICAL*4





APPENDIX B

DIAGNOSTIC MESSAGES

B.1 DIAGNOSTIC MESSAGES OVERVIEW

Diagnostic messages related to a VAX-11 FORTRAN IV-PLUS program can come from the compiler, the linker, or the Run-Time Library. The compiler detects syntax errors in the source program, such as unmatched parentheses, illegal characters, misspelled keywords, and missing or illegal parameters. The Run-Time Library reports errors that occur during execution. (Linker messages are summarized in the VAX-11 Linker Reference Manual.)

B.2 DIAGNOSTIC MESSAGES FROM THE COMPILER

The diagnostic messages issued by the compiler describe the error that has been detected, and in some cases contain an indication of the action taken by the compiler in response to the error.

Besides reporting errors detected in source program syntax, the compiler will issue messages indicating errors that involve the compiler itself, such as I/O errors, stack overflow, etc.

B.2.1 Source Program Diagnostic Messages

Three classes of source program errors are recognized, based on the severity of the error. These are (from greatest to least severity):

F Fatal; must be corrected before the program can be compiled. No object file is produced if an F-class error is detected during compilation.

E Error; not fatal but should be corrected. An object file is produced in spite of the presence of an E-class error, but the program will probably not execute properly.

W Warning; issued for statements that use acceptable, but non-standard syntax, and for statements corrected by the compiler. An object file is produced but you should check the statements to which a W-class diagnostic message applies, to make sure the program will produce the correct result. Note that W-class messages are produced unless the /NOWARNINGS qualifier is set in the FORTRAN command.

Typing mistakes are a likely cause of syntax errors, and can cause the compiler to generate misleading diagnostic messages. Beware especially of the following:

1. Missing comma or parenthesis in a complicated expression or FORMAT statement.

2. Misspelled variable names. The compiler may not detect this error, so execution can be affected.

3. Inadvertent line continuation mark. This can cause a diagnostic message for the preceding line.

4. Confusion between the digit zero (0) and the upper-case letter O, which can result in variable names that appear identical to you, but not to the compiler.

Another source of diagnostic messages is the inclusion of invalid ASCII characters in the source program. With the exception of the tab, space, and form-feed characters, non-printing ASCII control characters are not valid in a FORTRAN source program. As the source program is scanned, such invalid characters are replaced by a question mark (?). However, because ? cannot occur in a FORTRAN statement, a syntax error usually results.

Thus, because the message indicates only the immediate cause, you should always check the entire source statement carefully.

Figure B-1 shows the form of source program diagnostic messages as they are displayed at your terminal, in interactive mode. Figure B-2 shows how these messages appear in listings.

```
%FORT-W-ERROR 83, Extra comma in format list
        [FORMAT (I3,)] in module MORTGAGE at line 9

%FORT-F-ERROR 69, Undefined statement label
        [ 66] in module MORTGAGE at line 14

%FORT-I-ERRSUM, 2 Errors MORT.FOR.10
```

Figure B-1 Sample Diagnostic Messages (Terminal Format)

```
        C       PROGRAM TO CALCULATE MONTHLY MORTGAGE PAYMENTS

0001            PROGRAM MORTGAGE

0002            TYPE 10
0003    10      FORMAT (' ENTER AMOUNT OF MORTGAGE ')
0004            ACCEPT 20,IPV
0005    20      FORMAT (I6)

0006            TYPE 30
0007    30      FORMAT (' ENTER LENGTH OF MORTGAGE IN MONTHS ')
0008            ACCEPT 40,IMON
0009    40      FORMAT (I3,)
%FORT-W-ERROR 83, Extra comma in format list
        [FORMAT (I3,)] in module MORTGAGE at line 9


0010            TYPE 50
0011    50      FORMAT (' ENTER ANNUAL INTEREST RATE ')
0012            ACCEPT 60,YINT
0013    60      FORMAT (F6.4)
0014            GO TO 66
0015    65      YI = YINT/12            !GET MONTHLY RATE
0016            IMON = -IMON
0017            FIPV = IPV*YI
0018            YI = YI + 1
0019            FIMON = YI ** IMON
0020            FIMON = 1-FIMON
0021            FMNTHLY = FIPV/FIMON

0022            TYPE 70,FMNTHLY
0023    70      FORMAT (' MONTHLY PAYMENT EQUALS ', F7.3 )
0024            STOP
0025            END
%FORT-F-ERROR 69, Undefined statement label
        [ 66] in module MORTGAGE at line 14
```

Figure B-2 Sample Diagnostic Messages (Listing Format)

Table B-1
Source Program Diagnostic Messages

Message Number	Text/Meaning
1	F Statement too complex A statement is too complex to be compiled. It must be subdivided into two or more statements.
2	F Compiler expression stack overflow An expression is too complex or there are too many actual arguments in a subprogram reference. A maximum of 60 actual arguments can be compiled. You can subdivide a complex expression, or reduce the number of arguments.
3	F Too many named COMMON blocks Reduce the number of named COMMON blocks.
4	F Source line too long, compilation terminated A source line longer than 88 characters was encountered. NOTE: The compiler examines only the first 72 characters on a line.
5	E Too many continuation lines, remainder ignored Up to 99 continuation lines are permitted, as determined by the /CONTINUATIONS=n qualifier (default, 19).
6	F Syntax error in INCLUDE file specification The file name string is not acceptable (invalid syntax, invalid qualifier, undefined device, etc.).
7	F Open failure on INCLUDE file The specified file could not be opened, possibly due to an incorrect file specification, non-existent file, unmounted volume, or a protection violation.
8	F INCLUDE files nested too deeply Reduce the level of INCLUDE nesting or increase the number of continuation lines permitted. Each INCLUDE file requires space equivalent to two continuation lines.
9	W Invalid statement label ignored An improperly formed statement label (e.g., containing letters) has been detected in columns 1 - 5 of an initial line. The statement label is ignored.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
10	W Redundant continuation mark ignored A continuation mark was detected where an initial line is required. The continuation mark is ignored.
11	E Extra characters following a valid statement Superfluous text was found at the end of a syntactically correct statement. Check for typing or syntax errors.
12	E Overflow while converting constant or constant expression The specified value of a constant is too large or too small to be represented.
13	E Missing exponent after E or D A floating point constant was specified in E or D notation, but the exponent was omitted.
14	E Missing apostrophe in character constant A character constant must be enclosed by apostrophes.
15	E Zero-length string The length specified for a character, Hollerith, hexadecimal, octal or Radix-50 constant must not be zero.
16	E String constant truncated to maximum length A character or Hollerith constant can contain up to 255 characters. A Radix-50 constant can contain up to 12 characters.
17	E Count of Hollerith or Radix50 constant too large, reduced The value specified by the integer preceding the H or R is greater than the number of characters remaining in the source statement.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
18	E Invalid character used in constant An invalid character was detected in a constant. Valid characters are: Hexadecimal: 0 - 9, A - F, a - f Octal: 0 - 7 Radix-50: A - Z, 0 - 9, $, period, or space For Radix-50, a space is substituted for the invalid character. For hexadecimal and octal, the entire constant is set to zero.
20	F Missing variable or subprogram name A required variable name or subprogram name was not found.
21	F Missing constant A required constant was not found.
22	F Missing variable or constant An expression, or a term of an expression, has been omitted. Examples: WRITE ( ) DIST = *TIME
23	F Missing operator or delimiter symbol Two terms of an expression are not separated by an operator, or a punctuation mark (such as a comma) has been omitted. Examples: CIRCUM = 3.14 DIAM IF (I 10,20,30
24	F Missing statement label A required statement label reference was omitted.
25	F Missing keyword A required keyword, such as TO, was omitted from a statement such as ASSIGN 10 TO I.
26	F Non-integer expression where integer value required An expression that must be of type INTEGER was another data type.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
27	F Non-logical expression where logical value required An expression that must be of type LOGICAL was another data type.
28	F Arithmetic expression where character value required An expression that must be of type CHARACTER was another data type.
29	F Character expression where arithmetic value required An expression that must be arithmetic (INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or COMPLEX) was of type CHARACTER.
30	F Variable name, constant, or expression invalid in this context A quantity has been used incorrectly; for example, the name of a subprogram was used where an arithmetic expression is required.
31	F Operation not permissible on these data types An invalid operation was specified, such as .AND. of two real variables.
32	E Arguments incompatible with function, assumed user supplied A function reference was made, using a processor-defined function name, but the argument list does not agree in order, number, or type with the processor-defined function requirements. The function is assumed to be supplied by you as an EXTERNAL function.
33	F Subscripted reference to non-array variable A variable that is not defined as an array cannot appear with subscripts.

34	E Substring reference used in invalid context
	
	A substring reference has been detected to a variable or array that is not of type CHARACTER. Example:
	
	`REAL X(10)`
	`Y = X(J:K)`

35	F Number of subscripts does not match array declaration More or fewer dimensions are referenced than were declared for the array.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
36	E More than 7 dimensions specified, remainder ignored An array can be defined as having up to seven dimensions.
37	F Non-constant subscript where constant required Subscript and substring expressions used in DATA and EQUIVALENCE statements must be constants.
38	E Adjustable array bounds must be dummy arguments or in COMMON Variables specified in dimension declarator expressions must either be subprogram dummy arguments or appear in COMMON.
39	F Adjustable array used in invalid context A reference was made to an adjustable array in a context where such a reference is not allowed.
40	F Passed length character name used in invalid context A reference was made to a passed length character array or variable in a context where such reference is not allowed.
41	E Subscript or substring expression value out of bounds An array element beyond the specified dimensions, or a character substring outside the specified bounds, has been referenced.
42	E Lower bound greater than upper bound in array declaration The upper bound of a dimension declarator must be equal to or greater than the lower bound.
43	F Character substring limits out of order The first character position of a substring expression is greater than the last character position. Example: C(5:3)

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
44	F Invalid use of FUNCTION name in CALL statement A CALL statement referred to a subprogram name that was used as a character function; or a CALL statement that included alternate return specifiers referred to a name that was defined as a data type other than INTEGER*4. Examples: 1. IMPLICIT CHARACTER*10(C) CSCAL = CFUNC(X) CALL CFUNC(X) 2. REAL*4 TCB CALL TCB(Y,&100)
45	E %VAL, %REF, or %DESCR used in invalid context The argument list built-in functions (%VAL, %REF, %DESCR) cannot be used outside an actual argument list. Example: X = %REF(Y)
46	F Invalid argument to %VAL, %REF, %DESCR, or %LOC The argument specified for one of the built-in functions is not valid. For example: %VAL (3.5D0) - argument cannot be double precision, character, or complex. %LOC (X+Y) - argument must not be an expression.
47	F Alternate return label used in invalid context An alternate return argument was used in a function reference.
50	W Name longer than 15 characters A symbolic name has been truncated to 15 characters.
51	F Multiple declaration of name A name appears in two or more inconsistent declaration statements.
52	E Multiple declaration of data type for variable, first used A variable appears in more than one data type declaration statement. The first type declaration is used.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
53	E Syntax error in IMPLICIT statement Improper syntax was used in an IMPLICIT statement. Refer to the VAX-11 FORTRAN IV-PLUS Language Reference Manual for the syntax rules.
54	E Letter mentioned twice in IMPLICIT statement, last used A letter has been given an implicit data type more than once. The last data type given is used.
55	F Incorrect length modifier in declaration An unacceptable length has been specified in a data type declaration (see "Type Declaration" in the VAX-11 FORTRAN IV-PLUS Language Reference Manual). Example: `INTEGER PIPES*8`
56	F Left side of assignment must be variable or array element The symbolic name to which the value of an expression is assigned must be a variable, array element, or character substring reference.

57	E Length specified must match character function declaration
	
	The length specifications for all ENTRY names in a character function subprogram must be the same. Example:
	
	`CHARACTER*15 FUNCTION F`
	`CHARACTER*20 G`
	`ENTRY G`

58	F Inconsistent function data types
	
	All entry names in a function subprogram must be either character or numeric. Example:
	
	`CHARACTER*15 FUNCTION F`
	`REAL X`
	`ENTRY X`

60	F Undimensioned array or function definition out of order Either a statement function definition has been found among executable statements, or an assignment statement has been detected involving an array for which dimensions have not been given.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
61	E Statement out of order, statement ignored The order of statements was not as specified in the VAX-11 FORTRAN IV-PLUS Language Reference Manual. The statement found out of order is ignored.
62	E Statement not valid in this program unit, statement ignored A program unit contains a statement that is not allowed; for example, a BLOCK DATA subprogram contains an executable statement.
63	F Statement cannot appear in logical IF statement A logical IF statement must not contain a DO statement or another logical IF, IF THEN, ELSEIF, ELSE, ENDIF, or END statement.
64	W No path to this statement Program control cannot reach this statement. The statement is deleted. Example: 10 I=I+1 GO TO 10 STOP
65	E Missing END statement, END is assumed An END statement was missing at the end of the last input file, and has been inserted.
66	W Statement cannot be labeled, label ignored A label was placed on a statement that does not permit labels. The label is ignored.
67	E Inconsistent usage of statement label Labels of executable statements have been confused with labels of FORMAT statements. Example: GO TO 10 10 FORMAT (I5)
68	E Multiple definition of a statement label, second ignored The same label appears on more than one statement. The first occurrence of the label is used.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
69	F Undefined statement label Reference has been made to a statement label that is not defined in the program unit.
70	F DO and IF statements nested too deeply DO loops and block IF statements cannot be nested beyond 20 levels.

71	F DO or IF statement incorrectly nested
	
	One of the following conditions was found:
	
	1. A statement label specified in a DO statement has been used previously.
	
	Example:
	
	`10 I=I+1`
	`J=J+1`
	`DO 10 K=1,10`
	`.`
	`.`
	`.`
	
	2. A DO loop contains an incomplete DO loop or IF block.
	
	Examples:
	
	`DO 10 I=1,10`
	`J=J+1`
	`DO 20 K=1,10`
	`J=J+K`
	`10 CONTINUE`
	
	The start of the incomplete IF block can be a block IF, ELSEIF or ELSE statement.
	
	`DO 10 I=1,10`
	`J=J+I`
	`IF (J.GT.20) THEN`
	`J=J-1`
	`ELSE`
	`J=J+1`
	`10 CONTINUE`
	`ENDIF`

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning

72	F Invalid control structure using ELSE IF, ELSE, or ENDIF
	
	The order of ELSEIF, ELSE, or ENDIF statements is incorrect.
	
	ELSEIF, ELSE, and ENDIF statements cannot stand alone. ELSEIF and ELSE must be preceded by either a block IF statement or an ELSEIF statement. ENDIF must be preceded by either a block IF, ELSEIF, or ELSE statement.
	
	Examples:
	
	`DO 10 I=1,10`
	`  J=J+I`
	`  ELSEIF (J.LE.K) THEN`
	
	Error: ELSEIF preceded by a DO statement.
	
	`IF (J.LT.K) THEN`
	`  J=I+J`
	`ELSE`
	`  J=I-J`
	`ELSEIF (J.EQ.K) THEN`
	`ENDIF`
	
	Error: ELSEIF preceded by an ELSE statement.

73	F Unclosed DO loop or IF block
	
	The terminal statement of a DO loop or the ENDIF statement of an IF block was not found.
	
	Example:
	
	`DO 20 I=1,10`
	`X=Y`
	`END`

74	W Statement cannot terminate a DO loop The terminal statement of a DO loop cannot be a GO TO, arithmetic IF, RETURN, block IF, ELSE, ELSEIF, ENDIF, DO or END statement.
75	E ENTRY within DO loop or IF block, statement ignored An ENTRY statement is not allowed within the range of a DO loop or IF block.
76	W Assignment to DO variable within loop The control variable of a DO loop has been altered within the range of the DO statement.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
77	F Alternate return specifier invalid in FUNCTION subprogram The argument list of a FUNCTION declaration contains an asterisk, or a RETURN statement in a function subprogram specifies an alternate return. Examples: 1. INTEGER FUNCTION TCB(ARG,*,X) 2. FUNCTION IMAX . . . RETURN I+J END
78	E Alternate return omitted in SUBROUTINE or ENTRY statement An asterisk is missing in the argument list of a subroutine for which an alternate return is specified. For example: SUBROUTINE XYZ(A,B) . . . RETURN 1 Or: ENTRY MIN(Q,R) . . . RETURN I+4
80	E Format groups nested too deeply Format groups cannot be nested beyond eight levels.
81	E Unbalanced parentheses in format list The number of right parentheses does not match the number of left parentheses.
82	E Missing separator between format items A required separator character has been omitted between fields in a FORMAT statement.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
83	W Extra comma in format list Example: FORMAT (I4,)
84	E Constant in format item out of range A numeric value in a FORMAT statement exceeds the allowable range. Refer to the VAX-11 FORTRAN IV-PLUS Language Reference Manual.
85	E Format item contains meaningless character An invalid character or a syntax error was detected in a FORMAT statement.
86	E Format item cannot be signed A signed constant is valid only with the P format code.
87	E Missing number in format list Example: FORMAT (F6.)
88	E Extra number in format list Example: FORMAT (I4,3)
89	F Invalid I/O specification for this type of I/O statement A syntax error was found in the portion of an I/O statement that precedes the I/O list. Examples: TYPE (6),J WRITE 100,J
90	F Format specifier in error The format specifier in an I/O statement is invalid. It must be one of the following: • label of a FORMAT statement • * (list directed) • a run-time format specifier: variable, array element, or character substring reference.
91	E END= or ERR= specification given twice, first used Two instances of either END= or ERR= were found. Control is transferred to the location specified in the first occurrence.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
92	F Syntax error in I/O list Improper syntax was detected in an I/O list.
93	F Missing I/O list An I/O list was not present where required.
94	F Invalid I/O list element for input statement An input statement I/O list contains an invalid element, such as an expression or a constant.
95	F Duplicated keyword in OPEN/CLOSE statement Each keyword subparameter in an OPEN or CLOSE statement can be specified only once.
96	F UNIT= keyword missing in OPEN/CLOSE statement An OPEN or CLOSE statement must include the UNIT= subparameter.
97	F Incorrect keyword in CLOSE statement A CLOSE statement contains a keyword that is valid only in an OPEN statement.
100	E Number of names exceeds number of values in DATA statement The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. The remaining variables and/or array elements are not initialized.
101	E Number of values exceeds number of names in DATA statement The number of variables or array elements to be initialized must match the number of constants specified in a DATA statement. The remaining constant values are ignored.
102	E Invalid repeat count in DATA statement, count ignored The repeat count in a DATA statement is not an unsigned, nonzero integer constant. It is ignored.
103	E Constant size exceeds variable size in DATA statement A constant in a DATA statement is larger than its corresponding variable.

(continued on next page)

Table B-1 (Cont.)
Source Program Diagnostic Messages

Message Number	Text/Meaning
104	E Character name incorrectly initialized with numeric value Character data with a length greater than 1 was initialized with a numeric value in a DATA statement. Example: CHARACTER*4 A DATA A/14/
105	F Program storage requirements exceed addressable memory The storage space allocated to the variables and arrays of the program unit exceeds the addressing range of the machine.
106	F Variable inconsistently equivalenced to itself EQUIVALENCE statements specify inconsistent relationships between variables or array elements. Example: EQUIVALENCE (A(1),A(2))
107	F Invalid equivalence of two variables in COMMON Variables in COMMON cannot be equivalenced to each other.
108	F EQUIVALENCE statement incorrectly expands a COMMON block A COMMON block cannot be extended beyond its beginning by an EQUIVALENCE statement.
109	W Mixed numeric and character elements in COMMON A COMMON block must not contain both numeric and character data.
110	W Mixed numeric and character elements in EQUIVALENCE Numeric and character variable and array elements cannot be equivalenced to each other.
111	E Invalid initialization of variable not in COMMON An attempt was made, in a BLOCK DATA subprogram, to initialize a variable that is not in a common block.

B.2.2 Compiler-Fatal Diagnostic Messages

Conditions can be encountered of such severity that compilation must be terminated at once. These conditions are caused by hardware errors, software errors, and errors that require changing the source program. Table B-2 lists the diagnostic messages that report the occurrence of such compiler-fatal errors.

Table B-2
Compiler-Fatal Diagnostic Messages

Message Number	Text/Meaning
201	F Open error on work file on WK0:
202	F Open error on temp file on WK0: FORTRAN IV-PLUS creates a temporary work file and zero, one, or two temporary scratch files during the compilation process. The compiler was unable to open these required files. Possibly the volume was not mounted, space was not available on the volume, or a protection violation occurred.
203	F I/O error on work file on WK0:
204	F I/O error on temp file on WK0:
205	F I/O error on source file
206	F I/O error on object file
207	F I/O error on listing file I/O errors report either hardware I/O errors, or software error conditions such as an attempt to write on a write-protected volume.
208	F Compiler dynamic memory overflow Reduce the number of continuation lines allowed or reduce the INCLUDE file nesting depth.
209	F Compiler work file overflow A single program unit is too large to be compiled. Specify /WORK_FILES=3 or divide the program into smaller units.
210	F Compiler internal consistency check An internal consistency check has failed. This error should be reported to DIGITAL in a Software Performance Report.
211	F Compiler control stack overflow The compiler's control stack overflowed. Simplifying the FORTRAN source program will correct the problem.
212	F Branch displacement out of range A label referenced by a GO TO statement is beyond the displacement limits of a branch in the generated code. Modify the source program to reduce the distance between the GO TO and its reference. (See "Compiler Limits," Section B.2.3.)

B.2.3 Compiler Limits

There are limits to the size and complexity of a single VAX-11 FORTRAN IV-PLUS program unit. There are also limits on the complexity of FORTRAN statements. In some cases, the limits are readily described; see Table B-3. In other cases, however, the limits are not so easily defined.

For example, the compiler uses external work files to store the symbol table and a compressed representation of the source program. The number of work files is controlled by the /WORK_FILES qualifier: the maximum is 3, which provides space for approximately 2000 or more lines of source code in a typical FORTRAN program unit. If you run out of work file space, error 209 occurs.

In some cases, the limits can be adjusted by re-linking the compiler and modifying the limits to suit your needs. Table B-3 shows two values for such limits, in the form m(n), where m is the default limit and n is the maximum. Limits for which only one value is shown are not adjustable.

Table B-3
Compiler Limits

Language Element	Limit
DO and block-IF statement nesting (combined)	20 (many)
Actual arguments per CALL or function reference	60
Named COMMON blocks	44 (250)
Format group nesting	8
Labels in computed or assigned GOTO list	250
Parentheses nesting in expressions	40 (many)
INCLUDE file nesting	10
Continuation lines	99
FORTRAN source line length	88 characters
Symbolic name length	15 characters
Constants	
Character, Hollerith	255 characters
Radix-50	12 characters
Array dimensions	7

Error 212 occurs if a branch instruction and its target statement are too far apart. The compiler generates references to executable code within a single program unit by means of 8-bit and 16-bit relative displacements. Thus, a jump must not cross more than 32K bytes.

You will not reach this limit if a program unit results in no more than 32K bytes of generated code. You are more likely to reach the work file limit first.

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined at system generation.

B.3 RUN-TIME DIAGNOSTIC MESSAGES

Errors that occur during execution of your FORTRAN program are reported by diagnostic messages from the Run-Time Library. These messages may result from: hardware conditions; file system errors; errors detected by RMS; errors that occur during transfer of data between the program and an internal record; computations that cause overflow or underflow; incorrect calls to the Run-Time Library; and problems in array descriptions and conditions detected by the operating system. Refer to the VAX-11 Common Run-Time Procedure Library Reference Manual for more information.

B.3.1 Run-Time Library Diagnostic Message Presentation

Run-Time Library diagnostic messages are usually sent to either your terminal (interactive mode) or the log file (batch mode).

B.3.2 Run-Time Library Diagnostic Messages

Table B-4 lists the Run-Time library messages.

There is a HELP file available, which contains these error messages and descriptive text. Thus you can print the descriptive text at your terminal, on-line. For example, to print the text for a file system error (such as a rewind error), type in the following:

```
HELP ERROR FOR$_REWERR
```

The condition symbols corresponding to the message numbers are listed in Table 6-1. For more information about the run-time HELP file, see the VAX-11 Common Run-Time Procedure Library Reference Manual.

Table B-4
Run-Time Diagnostic Messages

Messages 20 through 48 indicate errors related to the file system. (No message numbers from 2 to 19 are used.)

Message Number	Text/Meaning
1	NOT A FORTRAN-SPECIFIC ERROR This message indicates that the Run-Time Library encountered an error that was not caused by a condition peculiar to FORTRAN. That is, it was a condition not described by any other message in this table.
20	REWIND ERROR An error condition was detected by RMS during an RMS$REWIND operation used to position a file at its beginning.
21	DUPLICATE FILE SPECIFICATIONS Multiple attempts were made to specify file attributes without an intervening close operation; i.e., DEFINEFILE followed by an OPEN statement or DEFINEFILE followed by DEFINEFILE.
22	INPUT RECORD TOO LONG Your program tried to read a record longer than the maximum record size. To read the file, use an OPEN statement with a RECORDSIZE value of the appropriate size.
23	BACKSPACE ERROR One of the following has occurred: 1. BACKSPACE was attempted on a file opened for appending 2. RMS detected an error condition during the RMS$REWIND operation used to rewind the file 3. RMS detected an error condition while reading forward to the desired record.
24	END-OF-FILE DURING READ Either an end-file record produced by the ENDFILE statement or the RMS end-of-file condition was encountered during a READ statement and no END= transfer specification was provided.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

Messages 20 through 48 indicate errors related to the file system. (No message numbers from 2 to 19 are used.)

Message Number	Text/Meaning
25	RECORD NUMBER OUTSIDE RANGE A direct access READ, WRITE, or FIND statement specified a record number outside the range specified in a DEFINEFILE statement or in the MAXREC keyword of the OPEN statement.
26	OPEN OR DEFINEFILE REQUIRED TO SPECIFY DIRECT ACCESS A direct access READ, WRITE or FIND operation was attempted before an OPEN statement specifying ACCESS='DIRECT', or DEFINEFILE statement was performed.
27	MORE THAN ONE RECORD IN I/O STATEMENT An attempt was made to read or write more than a single record in a direct access READ or WRITE statement, or an ENCODE or DECODE statement.
28	CLOSE ERROR An error was detected during an RMS$CLOSE, RMS$DELETE, or RMS$SPOOL operation, when RMS attempted to close the file.
29	FILE NOT FOUND A file with the specified name could not be found during an open operation.
30	OPEN FAILURE RMS detected an error condition during an open operation. (This message is used when the error condition is not one of the more common conditions for which specific error messages are provided.)
31	MIXED FILE ACCESS MODES An attempt was made to use both formatted and unformatted operations, or both sequential and direct access operations, on the same unit.
32	INVALID LOGICAL UNIT NUMBER A logical unit number greater than 99 or less than 0 was used.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

Messages 20 through 48 indicate errors related to the file system. (No message numbers from 2 to 19 are used.)

Message Number	Text/Meaning
33	ENDFILE ERROR An end-file record may not be written to a direct access file, a relative file, or to an unformatted sequential file that does not contain segmented records.
34	UNIT ALREADY OPEN An OPEN statement or DEFINEFILE statement was attempted that specified a logical unit already opened for input/output.
35	SEGMENTED RECORD FORMAT ERROR Invalid segmented record control data was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or written by a language other than FORTRAN.
36	ATTEMPT TO READ NON-EXISTENT RECORD One of the following has occurred: • A direct access read to a relative file has been attempted, specifying a record that was either never written, or deleted. • A direct access read has been attempted, specifying a record beyond end-of-file.
37	INCONSISTENT RECORD LENGTH An existing direct access file was opened whose record length attribute is not the same as specified in the OPEN or DEFINEFILE statement. It is possible the file was not created as a direct access file.
38	ERROR DURING WRITE RMS detected an error condition while writing.
39	ERROR DURING READ RMS detected an error condition while reading.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

Messages 20 through 48 indicate errors related to the file system. (No message numbers from 2 to 19 are used.)

Message Number	Text/Meaning
40	RECURSIVE I/O OPERATION I/O was attempted on a logical unit while that unit was involved in a preceding I/O operation. This can happen if a function that performs I/O to the same logical unit is referenced in an expression in an I/O list; or if a condition handler executes an I/O statement in response to an exception from an I/O statement for the same logical unit. Note that I/O can be performed to a different logical unit.
41	INSUFFICIENT VIRTUAL MEMORY The FORTRAN I/O library attempted to exceed its virtual page limit while dynamically allocating space for an I/O statement.
42	NO SUCH DEVICE A filename specification included an invalid device name when an open operation was attempted.
43	FILE NAME SPECIFICATION ERROR The filename string used in an OPEN statement is syntactically invalid, or is otherwise not acceptable to the operating system.
44	RECORD SPECIFICATION ERROR The RECORDSIZE value in an OPEN statement or the record size parameter in a DEFINEFILE statement is invalid (0 or negative) or is missing on an attempt to create a relative file or a file with fixed length records.
45	KEYWORD VALUE ERROR IN OPEN STATEMENT An OPEN statement keyword that requires a value has an invalid value. The following values are accepted: a. BLOCKSIZE: 0 to 65535 b. EXTENDSIZE: 0 to 65535 c. INITIALSIZE: 0 to 2**31-1 d. MAXREC 0 to 2**31-1 e. BUFFERCOUNT 0 to 127 f. RECORDSIZE 32766 for sequential organization 16380 for relative organization 9999 for variable length records on magnetic tape

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

Messages 20 through 48 indicate errors related to the file system. (No message numbers from 2 to 19 are used.)

Message Number	Text/Meaning
46	INCONSISTENT OPEN/CLOSE PARAMETERS The specifications in an OPEN and/or subsequent CLOSE statement indicated one or more of the following: a. A 'NEW' or 'SCRATCH' file which is 'READONLY' b. 'APPEND' to a 'NEW', 'SCRATCH', or 'READONLY' file c. 'SAVE' or 'PRINT' on a 'SCRATCH' file d. 'DELETE' or 'PRINT' on a 'READONLY' file.
47	WRITE TO READONLY FILE A write operation was attempted to a file declared to be READONLY by currently active OPEN.
48	INVALID ARGUMENT TO FORTRAN I/O LIBRARY A coded argument is not one of the defined set of codes on a call to the FORTRAN I/O library. This cannot occur in a FORTRAN I/O statement unless the version of the compiler is newer than the version of the Run-Time Library.
The following messages indicate errors in transmitting data between a FORTRAN program and an internal record.	
59	LIST-DIRECTED I/O SYNTAX ERROR The data in a list-directed input record has an invalid format, or the type of the constant is incompatible with the corresponding variable. The value of the variable is unchanged.

60	INFINITE FORMAT LOOP
	
	The format associated with an I/O statement that includes an I/O list has no field descriptors to use in transferring those variables.
	
	For example:
	
	`WRITE(1,1)X`
	`1 FORMAT('X=')`

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

The following messages indicate errors in transmitting data between a FORTRAN program and an internal record.

Message Number	Text/Meaning
61	FORMAT/VARIABLE-TYPE MISMATCH An attempt was made to input or output a real variable with an integer field descriptor (I, O, Z, or L), or an integer or logical variable with a real field descriptor (D, E, F, or G).
62	SYNTAX ERROR IN FORMAT A syntax error was encountered while the Run-Time Library was processing a format stored in an array.
63	OUTPUT CONVERSION ERROR During a formatted output operation the value of a particular number could not be output in the specified field length without loss of significant digits. The field is filled with asterisks.
64	INPUT CONVERSION ERROR During a formatted input operation an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable is set to zero.
66	OUTPUT STATEMENT OVERFLOWS RECORD An output statement specified an I/O list that exceeds the maximum record size specified. The record size is specified by the RECORDSIZE keyword of the OPEN statement, or by the record length attribute of an existing file.
67	INPUT STATEMENT REQUIRES TOO MUCH DATA A READ statement attempted to input more data than existed in the record being read. For example, the I/O list might have too many elements.
68	VARIABLE FORMAT EXPRESSION VALUE ERROR The value of a variable format expression is not within the range acceptable for its intended use; for example, a field width less than or equal to zero.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

The following messages indicate arithmetic overflow and underflow conditions.

Message Number	Text/Meaning
70	INTEGER OVERFLOW During an arithmetic operation an integer value exceeded BYTE, INTEGER*2 or INTEGER*4 range. The result of the operation is the correct low order part. This error will occur only for program units compiled with the /CHECK=OVERFLOW qualifier in effect.
71	INTEGER ZERO DIVIDE During an integer mode arithmetic operation an attempt was made to divide by zero. The result is set to the dividend, which is equivalent to division by 1.
72	FLOATING OVERFLOW During an arithmetic operation a value exceeded the largest representable real number. The result of the operation is set to minus 0.0.
73	FLOATING ZERO DIVIDE During a floating point arithmetic operation an attempt was made to divide by zero. The result of the operation is set to minus 0.0.
74	FLOATING UNDERFLOW During an arithmetic operation a floating point value has become less than the smallest representable real number, and has been replaced with a value of zero. This error is normally disabled. It may be enabled by calling LIB$FLT_UNDER.
77	SUBSCRIPT OUT OF RANGE An array reference has been detected that is outside the array as described by the array declarator. This checking is performed only for program units compiled with the /CHECK=BOUNDS qualifier in effect.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

The following messages result from incorrect calls to the Mathematical Procedures Library.

Message Number	Text/Meaning
80	WRONG NUMBER OF ARGUMENTS A library function was called with an improper number of arguments.
81	INVALID ARGUMENT TO MATH LIBRARY A math library function detected an invalid argument value.
82	UNDEFINED EXPONENTIATION Exponentiation was attempted that is mathematically undefined; e.g., 0.**0. The result is set to minus 0.0 for floating point operations, and 0 for integer operations.
83	LOGARITHM OF ZERO OR NEGATIVE VALUE An attempt was made to take the logarithm of zero or a negative number. The result is set to minus 0.0.
84	SQUARE ROOT OF NEGATIVE VALUE An argument required the evaluation of the square root of a negative value. The result is set to minus 0.0.
87	SINE OR COSINE SIGNIFICANCE LOST The magnitude of an argument to SIN or COS, or DSIN or DCOS was greater than 2**31 or 2**63 respectively, so that all significance was lost. The result is set to minus 0.0.
88	FLOATING OVERFLOW IN MATH LIBRARY An overflow condition was detected during execution of a mathematical library procedure. The result is set to minus 0.0.
89	FLOATING UNDERFLOW IN MATH LIBRARY An underflow condition was detected during execution of a mathematical library procedure. The result is set to minus 0.0.

(continued on next page)

Table B-4 (Cont.)
Run-Time Diagnostic Messages

The following message may be displayed if the dimensions of an adjustable array are inappropriately defined.

Message Number	Text/Meaning
93	ADJUSTABLE ARRAY DIMENSION ERROR Upon entry to a subprogram, the evaluation of dimensioning information has detected an array in which: • an upper dimension bound is less than a lower dimension bound, or • the dimensions imply an array that exceeds the addressable memory.

APPENDIX C

SYSTEM SUBROUTINES

C.1 SYSTEM SUBROUTINE SUMMARY

The VAX-11 FORTRAN IV-PLUS system provides subroutines you call in the same manner as a user-written subroutine. These subroutines are described in this Appendix.

The subroutines supplied are:

DATE	Returns a 9-byte string containing the ASCII representation of the current date.
IDATE	Returns three integer values representing the current month, day and year.
ERRSNS	Returns information about the most recently detected error condition.
EXIT	Terminates the execution of a program and returns control to the operating system.
SECNDS	Provides system time of day or elapsed time as a floating point value in seconds.
TIME	Returns an 8-byte string containing the ASCII representation of the current time in hours, minutes and seconds.

References to integer arguments in the following subroutine descriptions refer to arguments of either type INTEGER*4 or type INTEGER*2. However, the arguments must be either all INTEGER*4 or all INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines if their value is within the INTEGER*2 range.

C.2 DATE

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form:

```
CALL DATE(buf)
```

where buf is a 9-byte variable, array, array element or character substring.

The date is returned as a 9-byte ASCII character string of the form:

```
dd-mmm-yy
```

where

```
dd  is the 2-digit date
mmm is the 3-letter month specification
yy  is the last two digits of the year
```

C.3 IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. The call to IDATE has the form:

```
CALL IDATE(i,j,k)
```

If the current date were October 9, 1979 the values of the integer variables upon return would be:

```
i = 10
j = 9
k = 79
```

C.4 ERRSNS

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form:

```
CALL ERRSNS(fnum,rmssts,rmsstv,iunit,condval)
```

where

fnum

is an integer variable or array element into which is stored the most recent FORTRAN error number.

A 0 is returned if no error has occurred since the last call to ERRSNS or if no error has occurred since the start of execution.

rmssts

is an integer variable or array element into which is stored the RMS completion status code if the last error was an RMS I/O error.

rmsstv

is an integer variable or array element into which is stored the RMS status value if the last error was an RMS I/O error. This status value provides additional status information.

iunit

is an integer variable or array element into which is stored the logical unit number if the last error was an I/O error.

condval

is an integer variable or array element into which is stored the actual VAX-11 condition value. See Chapter 6 for a description of condition values.

Any of the arguments may be null. If the arguments are of INTEGER*2 type, only the low-order 16 bits of information are returned. The saved error information is set to zero after each call to ERRSNS.

C.5 EXIT

The EXIT subroutine causes program termination, closes all files, and returns control to the operating system. A call to EXIT has the form:

```
CALL EXIT [(exit-status)]
```

where (exit-status) is an optional integer argument you can use to specify the image exit status value.

C.6 SECNDS

The SECNDS function subprogram returns the system time in seconds as a single precision floating point value less the value of its single precision floating point argument. The call to SECNDS has the form:

```
y = SECNDS(x)
```

where y is set equal to the time in seconds since midnight minus the user-supplied value of x.

The SECNDS function can be used to perform elapsed time computations. For example:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)

C
C     CODE TO BE TIMED
C
      DELTA = SECNDS(T1)
```

where DELTA will give the elapsed time.

The value of SECNDS is accurate to 0.01 seconds, which is the resolution of the system clock.

Notes:

1. The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.

2. The 24 bits of precision provides accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

C.7 TIME

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form:

```
CALL TIME(buf)
```

where buf is an 8-byte variable, array, array element, or character substring.

The TIME call returns the time as an 8-byte ASCII character string of the form:

```
hh:mm:ss
```

where

```
hh is the 2-digit hour indication
mm is the 2-digit minute indication
ss is the 2-digit second indication
```

For example:

```
10:45:23
```

A 24-hour clock is used.

APPENDIX D

COMPATIBILITY

D.1 COMPATIBILITY: OVERVIEW

VAX-11 FORTRAN IV-PLUS is a compatible superset of PDP-11 FORTRAN IV and PDP-11 FORTRAN IV-PLUS. Generally speaking, any PDP-11 FORTRAN program will run correctly on VAX-11. Execution may be affected in some cases, however, because of differences in the hardware architecture of PDP-11 and VAX-11 computers, and differences between the IAS/RSX-11 and VAX/VMS operating environments.

The issues discussed in this appendix concern differences in language, run-time support, and utilities provided in the form of subroutines.

D.2 LANGUAGE DIFFERENCES

Differences related to language involve:

1. Logical tests
2. Floating point results
3. Character and Hollerith constants
4. Logical unit numbers
5. Assigned GO TO label list
6. Effect of DISPOSE='PRINT' specification

D.2.1 Logical Tests

The logical constants .TRUE. and .FALSE. are defined, respectively, as all 1's and all 0's by both VAX-11 FORTRAN IV-PLUS and PDP-11 FORTRAN. The test for .TRUE. and .FALSE. differs, however.

VAX-11 FORTRAN IV-PLUS tests the low-order bit (bit 0) of a logical value. This is the system-wide VAX-11 convention for testing logical values.

PDP-11 FORTRAN IV-PLUS tests the sign bit of a logical value; bit 7 for LOGICAL*1, bit 15 for LOGICAL*2, and bit 31 for LOGICAL*4.

PDP-11 FORTRAN IV tests the low-order byte of a logical value; all 0's is .FALSE., while any non-zero bit pattern is .TRUE.

In most cases, this difference will have no effect. It will be significant only for non-standard FORTRAN programs that perform arithmetic operations on logical values, and then make logical tests on the result.

Example:

```
LOGICAL*1 BA
BA=3
IF (BA) GO TO 10
```

VAX-11 FORTRAN IV-PLUS will produce a value of .TRUE., while PDP-11 FORTRAN IV-PLUS will produce .FALSE. PDP-11 FORTRAN IV will produce .TRUE.

D.2.2 Floating Point Results

Differences in math library routine results may occur because of new implementations of these routines, exploiting the VAX-11 instructions. The VAX-11 functions produce results of an accuracy equal to or greater than the corresponding PDP-11 functions, but there may be differences.

D.2.3 Character and Hollerith Constants

VAX-11 FORTRAN IV-PLUS supports both Hollerith and character constants, with the notations nHa...a and 'aaaa' respectively. In PDP-11 FORTRAN IV-PLUS, both notations are used for Hollerith constants. (Note that Hollerith constants have no data type, but assume a data type consistent with their use.)

In most cases, the conflicting use of the 'aaaa' notation is not a problem: VAX-11 FORTRAN IV-PLUS can determine from the program context whether a character or a Hollerith constant is intended. There is, however, one case in which this is not so. In an actual argument list for a CALL or function reference, where the subprogram called is a dummy argument, a constant in the 'aaaa' notation is always passed as a character constant, never as Hollerith. For example, given

```
SUBROUTINE S(F)
   .
   .
   .
CALL F('ABCD')
```

if the subroutine referenced by F expects a Hollerith constant (i.e., the dummy argument is numeric data type), execution will not be correct. The actual and dummy arguments must agree in data type. This will not be the case in the example shown. To avoid this problem, you must change to the nHaaa notation; as:

```
SUBROUTINE S(F)
   .
   .
   .
CALL F(4HABCD)
```

D.2.4 Logical Unit Numbers

If you do not specify a logical unit number in an I/O statement, a default unit number will be used. The defaults used by VAX-11 FORTRAN IV-PLUS differ from those used by PDP-11 FORTRAN IV-PLUS, as shown in Table D-1.

Table D-1
Default Logical Unit Numbers

I/O Statement	PDP-11 Unit	VAX-11 Unit
READ	1	-4
PRINT	6	-1
TYPE	5	-2
ACCEPT	5	-3

Note that PDP-11 FORTRAN IV-PLUS uses normal logical unit numbers, but VAX-11 FORTRAN IV-PLUS uses unit numbers that are not available to users. This prevents conflicts between these I/O statements and I/O statements that use explicit logical unit numbers. This should have no visible effect on program execution.

D.2.5 Assigned GO TO Label List

The labels specified in an assigned GO TO label list are checked by the VAX-11 FORTRAN IV-PLUS compiler to ensure their validity in the program unit. However, VAX-11 FORTRAN IV-PLUS does not perform a check at run time to ensure that a label actually assigned is in the list. PDP-11 FORTRAN IV-PLUS does perform this check at run time, but PDP-11 FORTRAN IV does not.

D.2.6 DISPOSE='PRINT' Specification

On some PDP-11 systems, the file is deleted after being printed if you specify DISPOSE='PRINT' in an OPEN or CLOSE statement. On VAX-11 FORTRAN IV-PLUS, the file is retained after being printed.

D.3 RUN-TIME SUPPORT DIFFERENCES

Differences in run-time support between VAX-11 FORTRAN IV-PLUS and PDP-11 FORTRAN are reflected in run-time error numbers, run-time error reporting, and in some OPEN statement keyword values.

D.3.1 Run-Time Library Error Numbers

Programs that use the ERRSNS subroutine may need to be modified because certain PDP-11 FORTRAN run-time error numbers have been either deleted from, or redefined in, the VAX-11 Run-Time Library. The error numbers affected are:

2 through 14	deleted; these error numbers reported fatal PDP-11 hardware conditions.
37 (INCONSISTENT RECORD LENGTH)	redefined; continuation is not allowed.
65 (FORMAT TOO BIG FOR 'FMTBUF')	deleted; this error cannot occur, because space is acquired dynamically for run-time formats.
72, 73, 82, 83, 84	redefined; floating point arithmetic errors and math library errors return -0.0 (a hardware reserved operand) rather than +0.0.
75 (FPP FLOATING TO INTEGER CONVERSION OVERFLOW)	deleted; error number 70 is reported instead.
86 (INVALID ERROR NUMBER)	deleted; error number 48 is reported instead.
91 (COMPUTED GOTO OUT OF RANGE)	deleted; no error is generated by the VAX-11 hardware when this condition occurs. Program execution continues in line.
92 (ASSIGNED LABEL NOT IN LIST)	deleted; as described in Section D.2.5, VAX-11 FORTRAN IV-PLUS does not perform this check at run-time.
94 (ARRAY REFERENCE OUTSIDE ARRAY)	deleted; error number 77 is reported instead.
95 through 101	deleted; these error numbers reported PDP-11 FORTRAN errors that cannot occur in VAX-11 FORTRAN IV-PLUS.

The following error numbers have been added:

35 (SEGMENTED RECORD FORMAT ERROR)

36 (ATTEMPT TO READ NON-EXISTENT RECORD)

48 (INVALID ARGUMENT TO FORTRAN I/O LIBRARY)

77 (SUBSCRIPT OUT OF RANGE)

87 (SINE OR COSINE SIGNIFICANCE LOST)

88 (FLOATING OVERFLOW IN MATH LIBRARY)

89 (FLOATING UNDERFLOW IN MATH LIBRARY)

See Table B-4 for descriptions of these error messages.

D.3.2 Error Handling and Reporting

VAX-11 FORTRAN IV-PLUS differs from PDP-11 FORTRAN IV-PLUS in the way it treats error continuation, I/O errors, and OPEN/CLOSE statement errors. Chapter 6 describes Run-Time Library error handling.

D.3.2.1 Continuing After Errors - In PDP-11 FORTRAN, program execution normally continues after errors such as floating overflow, until 15 such errors have occurred, at which point execution is terminated. VAX-11 FORTRAN IV-PLUS, however, sets a limit of 1 such error; program execution normally terminates when the first such error occurs. To change this behavior, you can take one of the following steps:

- Include a condition handler in your program to change the severity level of the error. Severity levels of Warning and Error permit continuation. See Chapter 6.
- Include the ERRSET subroutine (see Section D.4.3). ERRSET alters the Run-Time Library's default error processing to match the behavior of PDP-11 FORTRAN IV-PLUS.

D.3.2.2 I/O Errors with ERR= Specified

VAX-11 FORTRAN IV-PLUS neither generates an error message nor increments the image error count when an I/O error occurs, if an ERR= specification was included in the I/O statement. PDP-11 FORTRAN both reports the error and increments the task error count.

D.3.2.3 OPEN/CLOSE Statement Errors

Unlike PDP-11 FORTRAN, VAX-11 FORTRAN IV-PLUS reports only the first error encountered in an OPEN or CLOSE statement. PDP-11 FORTRAN reports all errors detected in processing the statement.

D.3.3 OPEN Statement Keywords

The space allocation for the INITIALSIZE keyword is contiguous best-try for VAX-11 FORTRAN IV-PLUS. That is, if you specify an INITIALSIZE value, and sufficient contiguous space is available, allocation will be contiguous. If there is not sufficient contiguous space, allocation will be non-contiguous. In PDP-11 FORTRAN IV-PLUS, allocation of contiguous or non-contiguous space depends on the sign of the value specified for the INITIALSIZE and EXTENDSIZE keywords. To be compatible with PDP-11 FORTRAN, VAX-11 FORTRAN IV-PLUS uses the absolute value of the user-supplied value.

D.4 UTILITY SUBROUTINES

There are a number of utility subroutines available for use with PDP-11 FORTRAN IV-PLUS. All are supplied as part of PDP-11 FORTRAN IV-PLUS, as described in the PDP-11 FORTRAN IV-PLUS User's Guide.

Six of these subroutines are supplied as a standard part of VAX-11 FORTRAN IV-PLUS (see Appendix C). These subroutines are:

```
DATE
ERRSNS
EXIT
IDATE
SECNDS
TIME
```

A new random number generator function is included in the Run-Time Library. For a description of this function, refer to the VAX-11 Common Run-Time Procedure Library Reference Manual.

The remaining subroutines are provided for purposes of compatibility: most have been superseded by features included in VAX-11 FORTRAN IV-PLUS, while others are of little applicability on VAX-11 systems. Sections D.4.1 through D.4.11 describe these routines.

The remaining utility subroutines are:

```
ASSIGN
CLOSE
ERRSET
ERRTST
FDBSET
IRAD50
RAD50
RAN
RANDU
R50ASC
USEREX
```

D.4.1 ASSIGN Subroutine

The ASSIGN subroutine is used to supply device or file name information for a logical unit. That is, it allows a device or file to be assigned to a logical unit. The assignment remains in effect until the program terminates or until the logical unit is closed by a CLOSE statement.

The ASSIGN subroutine must be called before the first I/O statement is issued for that logical unit.

The CALL FDBSET, CALL ASSIGN, and DEFINE FILE statements can be used together, but none can be used in conjunction with the OPEN statement for the same unit.

There are two other ways to assign a device or a file name to a logical unit number: specify the NAME keyword in an OPEN statement, or use the ASSIGN system command.

Format:

```
CALL ASSIGN (n[,name][,icnt])
```

Arguments:

n
: an integer value specifying the logical unit number

name
a variable, array, array element, or character constant containing any standard file specification

icnt
an INTEGER*2 value that specifies the number of characters contained in the string name

Note:

If only the unit number is specified, all previously specified file/device associations pertaining to that unit are nullified, and the defaults become effective. If icnt is omitted (or specified as zero), the file specification (if specified) is read until the first ASCII null character is encountered. If the icnt argument is specified, then the name argument must also be specified.

D.4.2 CLOSE Subroutine

The CLOSE subroutine closes the file currently open on a logical unit.

Format:

```
CALL CLOSE(n)
```

Argument:

n
an integer value specifying the logical unit

After the file is closed, the logical unit again assumes the default file name specification.

D.4.3 ERRSET Subroutine

The ERRSET subroutine determines the action taken in response to an error detected by the Run-Time Library. The VAX-11 Condition Handling Facility provides a more general method of defining actions to be taken when errors are detected (see Chapter 6).

Format:

```
CALL ERRSET(number, contin, count, type, log, maxlim)
```

Arguments:

number
an integer value specifying the error number

contin
a logical value:

.TRUE. - continue after error is detected

.FALSE. - exit after error is detected

count
a logical value:

.TRUE. - count the error against the maximum error limit

.FALSE. - do not count the error against the maximum error limit

type
a logical value:

.TRUE. - control passed to ERR= transfer label, if specified

.FALSE. - return to routine that detected the error, for default error recovery

NOTE

PDP-11 FORTRAN and VAX-11 FORTRAN differ in this respect: On PDP-11, this value takes precedence over an ERR= specification in the I/O statement; on VAX-11, the specification or omission of ERR= takes precedence over this value. That is, if ERR= was specified, control is transferred on any error, regardless of the value of type.

log
a logical value:

.TRUE. - produce an error message for this error

.FALSE. - do not produce an error message for this error

maxlim
positive INTEGER*2 value specifying the maximum error limit. The default is set to 15 at program initialization.

Notes:

1. The error action specified for each error is independent of other errors.

2. Null arguments are legal for all arguments except number, and have no effect on the current state of that argument.

3. An external reference to ERRSET or ERRTST causes a special PDP-11 FORTRAN compatibility error handler to be established before the main program is called. This special error handler transforms the executing environment to approximate that of PDP-11 FORTRAN.

D.4.4 ERRTST Subroutine

The ERRTST subroutine checks for a specific error. To perform appropriate actions in response to errors, you should establish a condition handler, as described in Chapter 6.

Format:

```
CALL ERRTST(i,j)
```

Arguments:

i
an integer value specifying the error number

j
a variable used for return value of error check

```
j = 1:     error i has occurred
j = 2:     error i has not occurred
```

Notes:

1. ERRTST resets the error flag for the specified error.

2. ERRTST is independent of the ERRSET subroutine. Neither subroutine has any direct effect on the other.

See also Note 3 under ERRSET.

Example:

```
     CALL ERRTST(43,J)
     GO TO (10,20)J
20   CONTINUE
```

If error 43 is detected, a branch is taken to statement 10 (J=1); if error 43 is not detected, control passes to statement 20 (J=2).

D.4.5 FDBSET Subroutine

The FDBSET subroutine is used to specify special I/O options. The recommended method of specifying I/O options is the OPEN statement.

Format:

```
CALL FDBSET(unit,acc,share,numbuf,initsz,extend)
```

Arguments:

unit
an integer value specifying the logical unit

acc
a character constant specifying the access mode to be used:

```
'READONLY'     read-only access
'NEW'          create a new file
'OLD'          access an existing file
'APPEND'       extend an existing sequential file
'UNKNOWN'      try 'OLD';  if no such file exists, use 'NEW'
```

share
a character constant 'SHARE' indicating that shared access is allowed

numbuf
an INTEGER*2 value specifying the number of buffers to be used for multibuffered I/O

initsz
an INTEGER*2 value specifying the number of blocks initially allocated for a new file

extend
an INTEGER*2 value specifying the number of blocks by which to extend a file

Notes:

1. FDBSET can be used only before issuing the first I/O statement for the unit.

2. CALL FDBSET, CALL ASSIGN, and the DEFINEFILE statement can be used together, but none can be used in conjunction with the OPEN statement for the same unit.

3. The unit argument must be specified. All other arguments are optional.

D.4.6 IRAD50 Subroutine

The IRAD50 subroutine is used to convert Hollerith data to Radix-50 form. IRAD50 may be called as a function subprogram if the return value is desired (format 1, below), or as a subroutine if the return value is not desired (format 2, below).

Formats:

1. n = IRAD50 (icnt,input,output)

2. CALL IRAD50(icnt,input,output)

Arguments:

n
(for function) an INTEGER*2 value indicating how many characters are converted

icnt
an INTEGER*2 value specifying the maximum number of characters to be converted

input
a Hollerith string to be converted to Radix-50

output
a numeric variable or array element where the Radix-50 results are stored

Notes:

1. Three Hollerith characters are packed into each output word. The number of output words is computed by the expression:

   (ICNT+2)/3

   Thus if a value of 4 is specified for icnt, two output words will result, even if an input string of only one character is converted.

2. Scanning of the input characters terminates on the first non-Radix-50 character in the input string.

D.4.7 RAD50 Function

The RAD50 function subprogram provides a simplified way to encode six Hollerith characters as two words of Radix-50 data.

Format:

```
RAD50(name)
```

Argument:

name
a numeric variable name or array element corresponding to a Hollerith string

Note: The RAD50 function is equivalent to:

```
FUNCTION RAD50(A)
CALL IRAD50(6,A,RAD50)
RETURN
END
```

D.4.8 RAN Function

The RAN function subprogram returns a pseudo-random number as the function value.

Format:

```
RAN(il,i2)
```

Arguments:

il,i2
INTEGER*2 variables or array elements that contain the seed for computing the random number.

Notes:

1. The values of il and i2 are updated during the computation to contain the updated seed.

2. The algorithm for computing the random number value is identical to the algorithm used in the RANDU subroutine (see Section D.4.9).

3. The RAN function is equivalent to:

```
FUNCTION RAN (I1,I2)
CALL RANDU (I1,I2,RAN)
RETURN
END
```

D.4.9 RANDU Subroutine

The RANDU subroutine computes a pseudo-random number, as a single precision value uniformly distributed in the range:

0.0 .LE. value .LT. 1.0

Format:

```
CALL RANDU(il,i2,x)
```

Arguments:

il,i2
INTEGER*2 variables or array elements that contain the seed for computing the random number.

x
a real variable or array element where the computed random number is stored.

Notes:

1. The values of il and i2 are updated during the computation to contain the updated seed.

2. The algorithm for computing the random number value is as follows:

   If I1=0, I2=0, set generator base

   X(n+1) = 2**16 + 3

   otherwise

   X(n+1) = (2**16+3) * X(n)mod 2**32

   Store generator base X(n+1) in I1,I2.

   Result is X(n+1) scaled to a real value Y(n+1), for 0.0 .LE. Y(n+1) .LT. 1.

D.4.10 R50ASC Subroutine

The R50ASC subprogram converts Radix-50 values to Hollerith strings.

Format:

```
CALL R50ASC(icnt,input,output)
```

Arguments:

icnt
INTEGER*2 value specifying the number of ASCII characters to be produced

input
numeric variable or array element containing the Radix-50 data

output
numeric variable or array element where the Hollerith characters are to be stored

Notes:

1. The number of words of input equals (icnt+2)/3.

2. If the undefined Radix-50 code is detected, or the Radix-50 word exceeds 174777 (octal), then question marks will be placed in the output location.

D.4.11 USEREX Subroutine

The USEREX subroutine specifies a routine to be called as part of the program termination process. This allows clean-up operations in non-FORTRAN routines.

You can establish a termination handler directly by calling the system service routine SYS$DCLEXH.

Format:

```
CALL USEREX(name)
```

Argument:

name
specifies the routine to be called

Notes:

1. The routine name must appear in an EXTERNAL statement in the program unit.

2. The user exit subroutine is called as a VAX/VMS termination handler. See the VAX/VMS System Services Reference Manual for information regarding termination handlers.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

VAX-11
FORTRAN IV-PLUS
User's Guide
AA-D035A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code_____
or
Country

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in tl
United State

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here