VAX FORTRAN User Manual

Order Number: AA-D035E-TE

June 1988

This manual describes how to compile, link, execute, and debug VAX FORTRAN programs on a VMS system. It also describes special features provided by VAX FORTRAN and a variety of system resources of interest to VAX FORTRAN programmers.

Revision/Update Information:	This revised manual supersedes the *VAX FORTRAN User's Guide* (order number AA-D035D-TE).
Operating System and Version:	VMS Version 5.0 or higher
Software Version:	VAX FORTRAN Version 5.0

digital equipment corporation
maynard, massachusetts

First printing, September 1984
Revised, June 1988

Contents

Preface

This manual explains how to create, compile, link, execute, and debug VAX FORTRAN programs on a VMS operating system.

This manual is designed to serve as a reference document, not as a tutorial document.

Intended Audience

This manual is intended for programmers and students who have a basic understanding of the FORTRAN language. It is not necessary for the reader to have a detailed understanding of the VMS operating system, but some familiarity with VMS is helpful. For detailed information concerning VMS or the VAX FORTRAN language, refer to the manuals listed under the heading Associated Documents.

Structure of This Document

This manual contains 15 chapters and 6 appendixes. The topics covered in the various chapters and appendixes are as follows:

- Chapter 1 describes the FORTRAN command line used to compile VAX FORTRAN programs.
- Chapter 2 describes how to link and run a VAX FORTRAN program.
- Chapter 3 describes the VMS Debugger and any special considerations involved in debugging VAX FORTRAN programs.

- Chapter 4 provides information on types and forms of I/O; file specifications, organizations, and access modes; environment variables and logical unit numbers; and I/O record formats.
- Chapter 5 describes how to control certain types of I/O errors using ERR, END, and IOSTAT specifiers in your I/O statements.
- Chapter 6 describes how to call routines and pass arguments to them.
- Chapter 7 describes how to utilize VMS Record Management Services (RMS) from a VAX FORTRAN program.
- Chapter 8 gives an introduction on how to exchange and share data among both local and remote processes.
- Chapter 9 describes facilities that can be used to handle—in a structured and consistent fashion—special conditions (errors or program-generated status conditions) that occur in large programs with many program units.
- Chapter 10 discusses a number of language features that may be implemented on VAX FORTRAN in a way that differs from other implementations of the FORTRAN language.
- Chapter 11 discusses source code optimizations performed by the VAX FORTRAN compiler.
- Chapter 12 describes how to use structures and records in VAX FORTRAN programs.
- Chapter 13 describes how to handle character data in VAX FORTRAN programs.
- Chapter 14 describes how to manipulate files using indexed sequential access.
- Chapter 15 describes the parallel processing support provided by VAX FORTRAN.
- Appendix A describes the debugging support for parallel processing.
- Appendix B identifies the VAX FORTRAN include files that define symbols for use in VAX FORTRAN programs.
- Appendix C contains examples of the use of a variety of system services.
- Appendix D details the differences between VAX FORTRAN and FORTRAN-66.
- Appendix E details the differences between VAX FORTRAN and PDP–11 FORTRAN.
- Appendix F explains the diagnostic messages that may be encountered by VAX FORTRAN programs.

Associated Documents

The *VAX FORTRAN Language Reference Manual* defines the format and use of statements in the VAX FORTRAN language.

The VMS documentation set provides detailed information on the VMS operating system.

Conventions Used in this Document

The following syntactic conventions are used in this manual:

- Uppercase words and letters used in examples indicate that you should type the word or letter as shown.
- Lowercase words and letters used in syntax specifications indicate that you are to substitute a word or value of your choice.
- Brackets ([ ]) indicate optional command elements.
- Braces ({ }) are used to enclose lists from which one command element is to be chosen.
- A horizontal ellipsis (. . .) indicates that the preceding item can be repeated one or more times.
- A vertical ellipsis in an example indicates that not all of the statements are shown.
- Conventions observed in references to typed data are as follows:
  - "Real" (lowercase) is used to refer to the REAL*4 (REAL), REAL*8, and REAL*16 data types as a group.
  - "Complex" (lowercase) is used to refer to the COMPLEX*8 (COMPLEX) and COMPLEX*16 (DOUBLE COMPLEX) data types as a group.
  - "Logical" (lowercase) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group.
  - "Integer" (lowercase) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.

In addition, the following notations are used to denote special nonprinting characters:

Tab character	<tab>
Space character	Δ

Summary of Changes

The documentation for VAX FORTRAN Version 5.0 is a major revision to the documentation provided for VAX FORTRAN Version 4.0.

The organization of the manual set has been changed. The manual set now comprises a language reference manual and a user manual.

- The language reference manual provides a detailed description of the VAX FORTRAN implementation of the FORTRAN language.
- The user manual describes how to compile, link, execute, and debug VAX FORTRAN programs. It also describes special features provided by VAX FORTRAN and a variety of system resources of interest to VAX FORTRAN programmers.

Major areas of change in the user manual are as follows:

- Many minor error corrections have been made throughout the entire manual.
- The chapters in the Version 4.0 user manual about the VMS operating system and the EDT editor have been removed. (Information on these topics can now be acquired only from the VMS documentation set.)
- The chapter on debugging has been entirely rewritten (and shortened).
- The chapter on calling conventions, "FORTRAN Call Conventions," has been revised, expanded, and renamed. Its new name is "Using VAX FORTRAN in the Common Language Environment."
- A new chapter, "VAX FORTRAN Support for Parallel Processing," has been added.
- A new appendix, "Working with the Multiprocess Debugging Configuration," has been added. (The information in this appendix is not available in the *VMS Debugger Manual* for VMS Version 5.0. It will be included in that manual when the manual is next revised or updated. To avoid duplication of information, the appendix will then be removed from the next version of the *VAX FORTRAN User Manual*.)

- New qualifiers for the FORTRAN command line are documented.
- New error messages covering "standards" (FORTRAN-77) checking and Version 5.0 language extensions are documented.

Chapter 1

Compiling VAX FORTRAN Programs

This chapter describes how to use the FORTRAN command to compile your source programs intp object modules. The following topics are discussed:

- The functions of the compiler (Section 1.1)
- The syntax of the FORTRAN command and its qualifiers (Section 1.2)
- The use of text libraries (Section 1.3)
- The Common Data Dictionary (CDD) (Section 1.4)
- Compiler diagnostic messages and error conditions (Section 1.5)
- Compiler output listing format (Section 1.6)

1.1 Functions of the Compiler

The primary functions of the VAX FORTRAN compiler are as follows:

- To verify the VAX FORTRAN source statements and to issue messages if the source statements contain any errors
- To generate machine language instructions from the source statements of the VAX FORTRAN program
- To group these instructions into an object module for the VMS linker

When the compiler creates an object file, it provides the linker with the following information:

- The program unit name. The program unit name is taken from the name specified in the PROGRAM, SUBROUTINE, FUNCTION, or

BLOCK DATA statement in the source program. If a program unit does not contain any of these statements, the source file name is used, with $MAIN (or $DATA, for block data subprograms) appended.

- A list of all entry points and common block names that are declared in the program unit. The linker uses this information when it binds two or more program units together and must resolve references to the same names in the program units.
- Traceback information. Traceback information is used by the system default condition handler when an error occurs that is not handled by the program itself. The traceback information permits the default handler to display a list of the active program units in the order of activation, which aids program debugging.
- A symbol table—if specifically requested (/DEBUG qualifier). A symbol table lists the names of all external and internal variables within a module, with definitions of their locations. The table is of primary use in program debugging.

The linker is described in Chapter 2.

1.2 The FORTRAN Command

The FORTRAN command initiates compilation of a source program.

The command has the following form:

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

/qualifiers
Indicates either special actions to be performed by the compiler or special properties of input or output files.

file-spec-list
Specifies the source files containing the program units 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 the command FORTRAN, followed by a carriage return. The system responds with the following prompt:

```
_File:
```

Type the file specification immediately after the prompt and then press the RETURN key.

1.2.1 Specifying Input Files

In specifying a list of input files on the FORTRAN command line, you can use abbreviated file specifications for those files that share common device names, directory names, or file names. The system applies temporary file specification defaults to those files with incomplete specifications. The defaults applied to an incomplete file specification are based on the previous device name, directory name, or file name encountered in the list.

For example, the following FORTRAN command line shows how temporary defaults are applied to a list of file specifications—given a current default device and directory name of USR2:[MONROE]:

```
$ FORTRAN USR1:[ADAMS]TEST1,TEST2,[JACKSON]SUMMARY,USR3:[FINAL]
```

The preceding FORTRAN command compiles the following files:

```
USR1:[ADAMS]TEST1.FOR
USR1:[ADAMS]TEST2.FOR
USR1:[JACKSON]SUMMARY.FOR
USR3:[FINAL]SUMMARY.FOR
```

To override a temporary default with your current default directory, specify the directory as a null value. For example:

```
$ FORTRAN [ALPHA]TEST1, []TEST2
```

In this case, the empty brackets indicate that the compiler is to use your current default directory to locate TEST2.

You must use the /LIBRARY qualifier in your FORTRAN command if text libraries are accessed by programs in the source files that you specify. The /LIBRARY qualifier is discussed at length in Section 1.3.3.

1.2.2 Specifying Output Files

The output produced by the compiler includes the object and listing files. You can control the production of these files by using the appropriate qualifiers on the FORTRAN command line.

The production of listing files depends on whether you are operating in interactive mode or batch mode:

- In interactive mode, the compiler does not generate listing files by default; you must use the /LIST qualifier to generate the listing file.
- In batch mode, the compiler generates a listing file by default. To suppress it, you must use the /NOLIST qualifier.

The compiler generates an object file by default. During the early stages of program development, you may find it helpful to use the /NOOBJECT qualifier to suppress the production of object files until your source program compiles without errors. 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 commas, each source file is compiled separately and an object file is generated for each source file.
- If you specify multiple source files, separated by plus signs, the source files are concatenated and compiled, and one object file is generated.

You can use both commas and plus signs in the same command line to produce different combinations of concatenated and separate object files (see Example 4 in this section).

To produce an object file with an explicit file specification, you must use the /OBJECT qualifier, in the form /OBJECT=file-spec (see Section 1.2.3.16). Otherwise, the object file has 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 fields (node, device, directory, and version) assume the default values.

The following examples show a variety of FORTRAN commands. Each command is followed by a description of the output files it produces.

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, but no listing file. (A listing file named XXX.LIS would be produced in batch mode.)

3.
```
$ FORTRAN/OBJECT=SQUARE/NOLIST RET
_File:  CIRCLE
```

   The source file CIRCLE.FOR is compiled, producing an object file named SQUARE.OBJ, but no listing file.

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 (comprising CCC.FOR).

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. The command shown in the previous example completely suppresses the object file. That is, source files ABC.FOR, CIRC.FOR, and XYZ.FOR are concatenated and compiled, but no object file is produced.

1.2.3 Qualifiers to the FORTRAN Command

FORTRAN command qualifiers influence the way in which the compiler processes a file. In many cases, the simplest form of the FORTRAN command is sufficient. However, you can select appropriate optional qualifiers if special processing is required.

Table 1–1 lists the FORTRAN command qualifiers. Sections 1.2.3.2 through 1.2.3.21 describe each qualifier in detail.

You can override some qualifiers specified on the command line by using the OPTIONS statement. The qualifiers specified by the OPTIONS statement affect only the program unit where the statement occurs. See the *VAX FORTRAN Language Reference Manual* for more information about the OPTIONS statement.

Table 1–1: FORTRAN Command Qualifiers

Qualifier	Negative Form	Default
/ANALYSIS_DATA[=filename]	/NOANALYSIS_DATA	/NOANALYSIS_DATA
/CHECK= { [NO]BOUNDS, [NO]OVERFLOW, [NO]UNDERFLOW, ALL, NONE }	/NOCHECK	CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
/CONTINUATIONS=n	None	/CONTINUATIONS=19
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
/DEBUG= { [NO]SYMBOLS, [NO]TRACEBACK, ALL, NONE }	/NODEBUG	/DEBUG=(NOSYMBOLS,TRACEBACK)
/D_LINES	/NOD_LINES	/NOD_LINES
/DIAGNOSTICS[=filename]	/NODIAGNOSTICS	/NODIAGNOSTICS
/DML	None	None
/EXTEND_SOURCE	/NOEXTEND_SOURCE	/NOEXTEND_SOURCE
/F77	/NOF77	/F77
/G_FLOATING	/NOG_FLOATING	/NOG_FLOATING
/I4	/NOI4	/I4
/LIBRARY	None	Not applicable

Table 1-1 (Cont.): FORTRAN Command Qualifiers

Qualifier	Negative Form	Default
/LIST[=file-spec]	/NOLIST	/NOLIST (interactive) /LIST (batch)
/MACHINE_CODE	/NOMACHINE_CODE	/NOMACHINE_CODE
/OBJECT[=file-spec]	/NOOBJECT	/OBJECT
/OPTIMIZE	/NOOPTIMIZE	/OPTIMIZE
/PARALLEL	/NOPARALLEL	/NOPARALLEL
/SHOW { [NO]DICTIONARY [NO]INCLUDE [NO]MAP [NO]PREPROCESSOR [NO]SINGLE ALL NONE }	/NOSHOW	/SHOW=(NODICTIONARY,NOINCLUDE,MAP, NOPREPROCESSOR,SINGLE)
/STANDARD= { [NO]SEMANTIC [NO]SOURCE_FORM [NO]SYNTAX ALL NONE }	/NOSTANDARD	/NOSTANDARD
/WARNINGS= { [NO]DECLARATIONS [NO]GENERAL [NO]ULTRIX [NO]VAXELN ALL NONE }	/NOWARNINGS	/WARNINGS=(NODECLARATIONS,GENERAL, NOULTRIX,NOVAXELN)

1.2.3.1 /ANALYSIS_DATA Qualifier

The /ANALYSIS_DATA qualifier produces a file that contains analysis data about the source code being compiled.

The qualifier has the following form:

```
/ANALYSIS_DATA=filename.type
```

The default file name is the name of the primary source file. The default file type is ANA (that is, filename.ANA).

If you do not specify the /ANALYSIS_DATA qualifier, the default behavior for the VAX FORTRAN compiler is /NOANALYSIS_DATA.

Source-code analysis files are reserved for use with DIGITAL products such as, but not limited to, the VAX Source Code Analyzer.

1.2.3.2 /CHECK Qualifier

The /CHECK qualifier produces run-time checks for the conditions indicated.

The qualifier has the following form:

```
          { ALL                                       }
/CHECK =  { ([NO]BOUNDS,[NO]OVERFLOW,[NO]UNDERFLOW) }
          { NONE                                      }
```

BOUNDS
Specifies that each dimension of an array reference or substring subscript reference is to be checked to determine whether it is within the range of the dimension specified by the array or character variable declaration. The default is NOBOUNDS.

OVERFLOW
Specifies that BYTE, INTEGER*2, and INTEGER*4 calculations are to be checked for arithmetic overflow. Real and complex calculations are always checked for overflow and are not affected by /NOCHECK. Integer exponentiation is performed by a routine in the mathematical library. The routine in the mathematical library always checks for overflow, even if /CHECK=NOOVERFLOW is specified.

UNDERFLOW
Specifies that real and complex calculations are to be checked for floating underflow. See Section 9.1.4.2 for information about floating underflow.

ALL
Specifies that OVERFLOW, BOUNDS, and UNDERFLOW checking is to be performed.

NONE
Specifies that no checking is to be performed.

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

1.2.3.3 /CONTINUATIONS Qualifier

The /CONTINUATIONS qualifier specifies the number of continuation lines allowed in a source program statement.

The qualifier has the following form:

```
/CONTINUATIONS=n
```

n
Is an integer from 0 to 99.

If you omit the /CONTINUATIONS qualifier, the default value is 19.

Because the compiler has to assume maximum-length continuation lines (66 or 126 characters) when allowing space for continuation-line sequences, the actual number of continuation lines allowed in any given statement usually exceeds the limit specified by the /CONTINUATIONS qualifier.

NOTE

A common problem is the attempt to use the character zero as a continuation indicator character in source code. This is not allowed. A line with a continuation indicator character of 0 is treated as an initial line; it does not cause an increase in the continuation value associated with the preceding line.

1.2.3.4 /CROSS_REFERENCE Qualifier

The /CROSS_REFERENCE qualifier specifies that the storage-map section of the listing file is to include information about the use of symbolic names. The cross-reference contains the numbers of the lines in which the symbols are defined and referenced.

The qualifier has the following form:

```
/CROSS_REFERENCE
```

The /CROSS_REFERENCE qualifier is ignored if the listing file is not being generated.

The default is /NOCROSS_REFERENCE.

See Section 1.6.3 for a description of the listing format used when /CROSS_REFERENCE is specified.

1.2.3.5 /DEBUG Qualifier

The /DEBUG qualifier specifies that the compiler is to provide information for use by the VMS Debugger and the run-time error traceback mechanism.

The qualifier has the following form:

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

SYMBOLS
Specifies that the compiler is to provide the debugger with local symbol definitions for user-defined variables, arrays (including dimension information), structures, parameter constants, and labels of executable statements.

TRACEBACK
Specifies that the compiler is to provide an address correlation table so that the debugger and the run-time error traceback mechanism can translate virtual addresses into source program routine names and compiler-generated line numbers.

ALL
Specifies that the compiler is to provide both local symbol definitions and an address correlation table.

NONE
Specifies that the compiler is to provide no debugging information.

The default is /DEBUG=TRACEBACK if you do not specify the /DEBUG qualifier. Note that /DEBUG is the equivalent of /DEBUG=ALL, and /NODEBUG is the equivalent of /DEBUG=NONE.

NOTE

> The use of /NOOPTIMIZE is strongly recommended when the /DEBUG qualifier is used. Optimizations performed by the compiler can cause several different kinds of unexpected behavior when using the VMS Debugger. See Section 11.2 for more information on this subject.

For more information on debugging and traceback, see Section 2.3 and Chapter 3.

1.2.3.6 /D_LINES Qualifier

The /D_LINES qualifier specifies that lines with a D in column 1 are to be compiled and are not to be treated as comment lines.

The qualifier has the following form:

```
/D_LINES
```

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

1.2.3.7 /DIAGNOSTICS Qualifier

The /DIAGNOSTICS qualfier creates a file containing compiler messages and diagnostic information.

The qualifier has the following form:

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

The default is /NODIAGNOSTICS.

If you omit the file specification, the diagnostics file defaults to the name of your source file (with a file type of DIA).

The diagnostics file is reserved for use with DIGITAL layered products such as the VAX Language-Sensitive Editor.

1.2.3.8 /DML Qualifier

The /DML qualifier specifies that the FORTRAN Data Manipulation Language (DML) preprocessor is to be invoked before the compiler. The preprocessor produces an intermediate file of VAX FORTRAN source code in which FORTRAN DML commands are expanded into VAX FORTRAN statements. The compiler is then automatically invoked to compile this intermediate file.

The qualifier has the following form:

```
/DML
```

Use the /SHOW=PREPROCESSOR qualifier in conjunction with the /DML qualifier to cause the preprocessor-generated source code to be included in the listing file. For more information on the DML preprocessor, refer to the *VAX DBMS FDML Reference Manual.*

NOTE

Because the intermediate file is deleted by the FORTRAN DML preprocessor immediately after compilation is complete, the Language Sensitive Editor and the Source Code Analyzer cannot access the source program when the /DML qualifier is used.

1.2.3.9 /EXTEND_SOURCE Qualifier

The /EXTEND_SOURCE qualifier specifies that the compiler is to extend the length of VAX FORTRAN statement fields to column 132, instead of column 72 (the default).

The qualifier has the following form:

```
/EXTEND_SOURCE
```

This qualifier can also be specified on the OPTIONS statement. The default in either case is /NOEXTEND_SOURCE.

If a source line is longer than 132 characters, a fatal read error is signaled and the compilation is terminated.

1.2.3.10 /F77 Qualifier

The /F77 qualifier specifies that FORTRAN–77 interpretation rules are used for those statements that have a meaning incompatible with FORTRAN-66. See Appendix D for a discussion of these incompatibilities.

The qualifier has the following form:

```
/F77
```

The default is /F77. If you specify /NOF77, the compiler selects FORTRAN-66 interpretations in cases of incompatibility.

1.2.3.11 /G_FLOATING Qualifier

The /G_FLOATING qualifier controls how the compiler implements REAL*8, COMPLEX*16, DOUBLE PRECISION, and DOUBLE COMPLEX quantities.

The qualifier has the following form:

```
/G_FLOATING
```

The /NOG_FLOATING qualifier, the default, causes the compiler to implement double-precision quantities using the VAX D_floating data type. The /G_FLOATING qualifier causes the compiler to implement such quantities using the VAX G_floating data type.

If your program requires the G_floating form of double precision for its correct operation (that is, it uses a range larger than 10**38), you should use the /G_FLOATING qualifier in an OPTIONS statement in your source program. The implementation of REAL*8 in VAX FORTRAN is further discussed in the *VAX FORTRAN Language Reference Manual.*

Note that you should not mix the D_floating and G_floating data types in routines that pass double-precision quantities between themselves.

NOTE

> VMS systems support both D_floating and G_floating implementations of the REAL*8 data type. On different systems, however, the performance of a program can vary widely, depending on whether your program is compiled with the G_floating option in effect or the D_floating option in effect. The disparity exists when a particular system supports one floating type in hardware and the other supports it in software.
>
> Thus, if you wish to optimize performance and if range and accuracy constraints do not disallow the use of either of the options, you must determine which implementation (D_floating or G_floating) is most efficient on the system you will be running your program on and then compile your program using the appropriate compilation option.
>
> You can select G_floating or D_floating by means of an OPTIONS statement in your source program or by means of qualifiers on the FORTRAN command line.

See the *VAX FORTRAN Language Reference Manual* for more information on floating-point data types.

1.2.3.12 /I4 Qualifier

The /I4 qualifier controls how the compiler interprets INTEGER and LOGICAL declarations that do not have a specified length.

The qualifier has the following 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.

1.2.3.13 /LIBRARY Qualifier

The /LIBRARY qualifier specifies that a file is a text library file.

The qualifier has the following form:

```
text-library-file/LIBRARY
```

The /LIBRARY qualifier can be specified on one or more text library files in a list of files concatenated by plus signs (+). At least one of the files in the list must be a nonlibrary file. The default file type is TLB.

The use of text libraries is discussed at length in Section 3.3.

1.2.3.14 /LIST Qualifier

The /LIST qualifier specifies that a source listing file is to be produced.

The qualifier has the following 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 to a file type of LIS.

In interactive mode, the compiler does not produce a listing file 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 1.6.1 for a discussion on the format of listing files.

1.2.3.15 /MACHINE_CODE Qualifier

The /MACHINE_CODE qualifier specifies that the listing file is to include a symbolic representation of the object code generated by the compiler. Generated code and data are represented in a form similar to a VAX MACRO assembly listing. Do not attempt to assemble this listing file; several items included in the listing file are not supported by VAX MACRO assembler.

The qualifier has the following form:

```
/MACHINE_CODE
```

This qualifier is ignored if no listing file is being generated. The default is /NOMACHINE_CODE.

See Section 1.6.2 for a description of the format of a machine code listing.

1.2.3.16 /OBJECT Qualifier

The /OBJECT qualifier specifies the name of the object file.

The qualifier has the following form:

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

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

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

1.2.3.17 /OPTIMIZE Qualifier

The /OPTIMIZE qualifier specifies that the compiler is to produce optimized code.

The qualifier has the following form:

```
/OPTIMIZE
```

The default is /OPTIMIZE. The negative form /NOOPTIMIZE should be used during a debugging session to ensure that the debugger has sufficient information to locate errors in the source program. (See Chapter 11 for information on optimizations performed by the VAX FORTRAN compiler.)

1.2.3.18 /PARALLEL Qualifier

The /PARALLEL qualifier controls whether special processing to support parallel processing is performed during compilation.

The qualifier has the following form:

```
/PARALLEL
```

The default is /NOPARALLEL. See Chapter 15, especially Section 15.7.1, for details on the use of the /PARALLEL qualifier.

1.2.3.19 /SHOW Qualifier

The /SHOW qualifier controls whether optionally listed source lines and a symbol map are to appear in the source listing. (Optionally listed source lines are text-module source lines and preprocessor-generated source lines.)

The /LIST qualifier must be specified in order for the /SHOW qualifier to take effect.

The qualifier has the following form:

```
        { ALL                                                                  }
/SHOW = { ([NO]DICTIONARY,[NO]INCLUDE,[NO]MAP,[NO]PREPROCESSOR,[NO]SINGLE) }
        { NONE                                                                 }
```

ALL

Specifies that all optionally listed source lines are to be included in the listing file.

INCLUDE

Specifies that the source lines from any file or text module specified by INCLUDE statements are to be included in the source listing.

DICTIONARY

Specifies that VAX FORTRAN source representations of any CDD records referenced by DICTIONARY statements are to be included in the listing file.

MAP

Specifies that the symbol map is to be included in the listing file. If the /CROSS_REFERENCE qualifier is specified, MAP is ignored.

PREPROCESSOR
Specifies that preprocessor-generated source lines are to be included in the listing file. The negative form, NOPREPROCESSOR, specifies that the preprocessor-generated source lines are to be excluded from the source listing.

SINGLE
Specifies that symbolic names of parameter constants are to be included in cross-reference listings—even if they are not referenced outside the PARAMETER statements in which they are declared. The negative form, NOSINGLE, specifies that names of parameter constants are to be suppressed if they are only declared and not referenced elsewhere. This is useful for cross-reference listings of small programs that specify INCLUDE declarations but use only a small number of the parameter constant names that have been declared.

NONE
Specifies that no optionally listed source lines are to be included in the listing file.

The /SHOW qualifier defaults are NOPREPROCESSOR, NOINCLUDE, NODICTIONARY, MAP, SINGLE.

Specifying the qualifier /SHOW without any arguments is equivalent to specifying /SHOW=ALL; specifying /NOSHOW without any arguments is equivalent to specifying /SHOW=NONE.

1.2.3.20 /STANDARD Qualifier

The /STANDARD qualifier specifies that the compiler is to generate informational diagnostics for VAX extensions to FORTRAN–77 that can be determined at compile-time.

The qualifier has the following form:

```
             { ALL                                        }
/STANDARD =  { ([NO]SEMANTIC,[NO]SOURCE_FORM,[NO]SYNTAX) }
             { NONE                                       }
```

SEMANTIC
Specifies that an informational message is to be issued for ANSI standard conforming statements that become nonstandard due to the way in which they are used. Data type information and statement locations are considered when determining semantic extensions. Specifying SEMANTIC checking also enables SYNTAX checking to be performed.

SOURCE_FORM
Specifies that an informational message is to be issued for statements that use tab formatting or contain lowercase characters.

SYNTAX
Specifies that an informational message is to be issued for syntax extensions to the current ANSI standard. SYNTAX extensions include nonstandard statements and languages constructs.

ALL
Specifies that informational messages are to be issued for semantic, source form, and syntax extensions to the current ANSI standard.

NONE
Specifies that no informational messages are to be issued for extensions to the current ANSI standard.

The default is /NOSTANDARD, which is equivalent to /STANDARD=NONE.

If you specify the /NOWARNINGS qualifier, the /STANDARD qualifier is ignored. Specifying /STANDARD with no arguments is equivalent to specifying /STANDARD=(SEMANTIC, NOSOURCE_FORM, SYNTAX).

1.2.3.21 /WARNINGS Qualifier

The /WARNINGS qualifier specifies that the compiler is to generate informational (I) and warning (W) diagnostic messages in response to informational and warning-level errors.

The qualifier has the following form:

```
             { ALL                                                  }
/WARNINGS = { ([NO]DECLARATIONS,[NO]GENERAL,[NO]ULTRIX,[NO]VAXELN) }
             { NONE                                                 }
```

DECLARATIONS
Causes the compiler to print warnings for any untyped data item used in the progran. DECLARATIONS acts as an external IMPLICIT NONE declaration. The default is NODECLARATIONS. See the description of the IMPLICIT statement in the *VAX FORTRAN Language Reference Manual* for information about the effects of IMPLICIT NONE.

GENERAL
Causes the compiler to generate informational and warning diagnostic messages. An informational message (I) indicates that a correct VAX FORTRAN statement may have unexpected results or contains nonstandard syntax or source form. A warning message (W) 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 informational and warning diagnostic messages, specify the negative form of this qualifier (NOGENERAL). The default is GENERAL.

ULTRIX
Causes the compiler to issue diagnostics for language features not supported by VAX FORTRAN on ULTRIX systems. Through the use of this option, you can develop VAX FORTRAN programs on a VMS system and use those programs—without modification—on both ULTRIX and VMS systems. The default is NOULTRIX.

VAXELN
Causes the compiler to issue diagnostic messages for language features not supported by VAX FORTRAN on a VAXELN system. The default is NOVAXELN.

ALL
Causes the compiler to print all informational and warning messages, including warning messages for any untyped data items and for language features not supported on ULTRIX or VAXELN. Specifying ALL has the effect of specifying (DECLARATIONS, GENERAL, ULTRIX, VAXELN).

NONE
Suppresses all informational and warning messages.

Appendix B discusses compiler diagnostic messages.

1.3 Using Text Libraries

A text library contains modules of source text that you can incorporate in a program by using the INCLUDE statement. Modules within a text library are like ordinary text files, but they differ in the following ways:

- They contain a unique name, called the module name, that is used to access them.
- Several can be contained within the same library file.

Modules in text libraries can contain any kind of text; this section only discusses their use when VAX FORTRAN language source is used.

Use the LIBRARY command of the VMS Librarian Utility to create and modify modules in text libraries. Text libraries have a default file type of TLB.

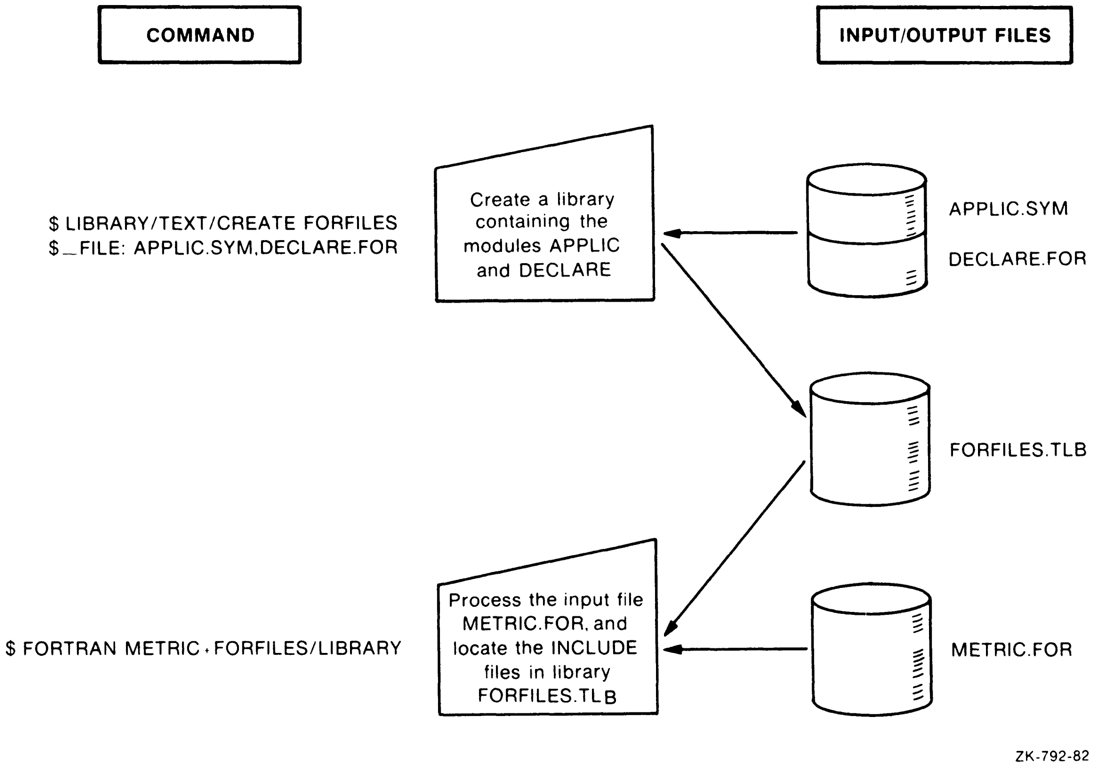
Use one of the following methods to access a source module in a text library.

- Specify only the name of the module in an INCLUDE statement in your VAX FORTRAN source program.
- Specify the name of both the library and module in an INCLUDE statement in your VAX FORTRAN source program.
- Define a default library using the logical name FORT$LIBRARY (see Section 1.3.4.1).
- Specify the name of the library using the /LIBRARY qualifier on the FORTRAN command line that you use to compile the source program (see Section 1.3.3).

For information on how to use INCLUDE statements, see the *VAX FORTRAN Language Reference Manual.*

Figure 1–1 shows the creation of a text library and its use in compiling VAX FORTRAN programs.

Figure 1: Creating and Using a Text Library


COMMAND
INPUT/OUTPUT FILES
$ LIBRARY/TEXT/CREATE FORFILES
$_FILE: APPLIC.SYM,DECLARE.FOR
Create a library containing the modules APPLIC and DECLARE
APPLIC.SYM
DECLARE.FOR
FORFILES.TLB
$ FORTRAN METRIC+FORFILES/LIBRARY
Process the input file METRIC.FOR, and locate the INCLUDE files in library FORFILES.TLB
METRIC.FOR
ZK-792-82

1.3.1 Using the LIBRARY Commands

Table 1–2 summarizes the commands that create libraries and provide maintenance functions. For a complete list of the qualifiers for the LIBRARY command and a description of other DIGITAL Command Language (DCL) commands listed in Table 1–2, see the *Guide to Using VMS Command Procedures.*

Table 1–2: Commands to Control Library Files

Function	Command Syntax[1]
Create a library	LIBRARY/TEXT/CREATE library-name file-spec,...
Add one or more modules to a library	LIBRARY/TEXT/INSERT library-name file-spec,...
Replace one or more modules in a library	LIBRARY/TEXT/REPLACE library-name file-spec,...[2]
Specify the names of modules to be added to a library	LIBRARY/TEXT/INSERT library-name file-spec/MODULE= module-name
Delete one or more modules from a library	LIBRARY/TEXT/DELETE=(module-name,...) library-name
Copy a module from a library into another file	LIBRARY/TEXT/EXTRACT=module-name/OUTPUT= file-spec library-name
List the modules in a library	LIBRARY/TEXT/LIST=file-spec library-name

[1]The LIBRARY command qualifier /TEXT indicates a text module library. By default, the LIBRARY command assumes an object module library.

[2]REPLACE is the default function of the LIBRARY command if no other action qualifiers are specified. If no module exists with the given name, /REPLACE is effectively /INSERT.

1.3.2 Naming Text Modules

When the LIBRARY command adds a module to a library, it uses by default the file name of the input file as the name of the module. In the example in Figure 1–1, the LIBRARY command adds the contents of the files APPLIC.SYM and DECLARE.FOR to the library and names the modules APPLIC and DECLARE.

Alternatively, you can name a module in a library with the /MODULE qualifier. For example:

```
$ LIBRARY/TEXT/INSERT FORFILES  DECLARE.FOR/MODULE=EXTERNAL_DECLARATIONS
```

The preceding command inserts the contents of the file DECLARE.FOR into the library FORFILES under the name EXTERNAL_DECLARATIONS. This module can be included in a VAX FORTRAN source file during compilation with the following statement:

```
INCLUDE 'FORFILES(EXTERNAL_DECLARATIONS)'
```

1.3.3 Specifying Library Files on the FORTRAN Command Line

The /LIBRARY qualifier is used on the FORTRAN command line to identify text libraries. If a source file that you are compiling includes a module from a text library, you concatenate the name of the text library to the name of the source file and append the /LIBRARY qualifier to the text library name. Concatenation is specified with a plus sign (+). For example:

```
$ FORTRAN APPLIC+DATAB/LIBRARY
```

Whenever an INCLUDE statement occurs in APPLIC.FOR, the compiler searches the library DATAB.TLB for the source text module identified in the INCLUDE statement and incorporates it into the compilation. See the *VAX FORTRAN Language Reference Manual* for a description of the INCLUDE statement.

1.3.4 Search Order of Libraries

When more than one library is specified on a FORTRAN command line, the VAX FORTRAN compiler searches the libraries each time it processes an INCLUDE statement that specifies a text module name. The compiler searches the libraries in the order specified on the command line. For example:

```
$ FORTRAN APPLIC+DATAB/LIBRARY+NAMES/LIBRARY+GLOBALSYMS/LIBRARY
```

When the VAX FORTRAN compiler processes an INCLUDE statement in the source file APPLIC.FOR, it searches the libraries DATAB.TLB, NAMES.TLB, and GLOBALSYMS.TLB, in that order, for source text modules identified in the INCLUDE statement.

On a command that requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

```
$ FORTRAN METRIC+DATAB/LIBRARY, APPLIC+DATAB/LIBRARY
```

In this example, VAX FORTRAN compiles METRIC.FOR and APPLIC.FOR separately and uses the library DATAB.TLB for each compilation.

After the compiler has searched all libraries specified on the command line, it searches the user-supplied default library, if any, specified by the logical name FORT$LIBRARY. Then, it searches the system-supplied default library SYS$LIBRARY:FORSYSDEF.TLB.

1.3.4.1 User-Supplied Default Libraries

You can define one of your private text libraries as a default library for the VAX FORTRAN compiler to search. The VAX FORTRAN compiler searches the default library after it searches libraries specified in the FORTRAN command.

To define a default library, assign an equivalence for the logical name FORT$LIBRARY, as in the following example of the DCL command ASSIGN.

```
$ ASSIGN DBA0:[LIB]DATAB FORT$LIBRARY
```

While this assignment is in effect, the compiler automatically searches the library DBA0:[LIB]DATAB.TLB for any include modules that it cannot locate in libraries explicitly specified, if any, on the FORTRAN command line.

You can define the logical name FORT$LIBRARY in any logical name table. If the name is defined in more than one table, the VAX FORTRAN compiler uses the equivalence for the first match it finds in the normal order of search—first the process table, then the group table, and finally the system table. Thus, if FORT$LIBRARY is defined in both the process and group logical name tables, the process logical name table assignment overrides the group logical name table assignment.

If FORT$LIBRARY is defined as a search list, the compiler opens the first text library specified in the list. If the include module is not found in that text library, the search is terminated and an error message is issued.

1.3.4.2 System-Supplied Default Library

When the VAX FORTRAN compiler cannot find the include modules in libraries specified on the FORTRAN command line or in the default library defined by FORT$LIBRARY, it then searches the system-supplied library SYS$LIBRARY:FORSYSDEF.TLB.

SYS$LIBRARY identifies the device and directory containing system libraries and is normally defined by the system manager. FORSYSDEF.TLB is a library of include modules supplied by VAX FORTRAN. It contains local symbol definitions and structures required for use with system services and return status values from system services.

Refer to Appendix B for more information on the contents of FORSYSDEF.

1.4 Using the VAX Common Data Dictionary

The Common Data Dictionary (CDD) is an optional VAX software product available under a separate license. The CDD allows you to maintain a set of shareable data definitions (language-independent structure declarations) that are defined by a system manager or data administrator. See the *VAX Common Data Dictionary Utilities Reference Manual* and the *VAX CDD Data Definition Language Reference Manual* for detailed information about the CDD.

CDD data definitions are organized hierarchically in much the same way that files are organized in directories and subdirectories. For example, a dictionary for defining personnel data might have separate directories for each employee type. A directory for salesmen might have subdirectories that include data definitions for records such as salary and commission history or personnel history.

Descriptions of data definitions are entered into the dictionary in a special-purpose language called CDDL (Common Data Dictionary Language). The CDDL compiler converts the data descriptions to an internal form—thus making them independent of the language used to access them—and inserts them into the CDD.

During the compilation of a VAX FORTRAN program, CDD data definitions can be accessed by means of DICTIONARY statements. If the data attributes of the data definitions are consistent with VAX FORTRAN requirements, the data definitions are included in the VAX FORTRAN program. CDD data definitions, in the form of VAX FORTRAN source code, appear in source program listings if you specify the

/SHOW=DICTIONARY qualifier on the FORTRAN command line or the /LIST qualifier in the DICTIONARY statement.

The advantage in using the CDD, instead of VAX FORTRAN source, for structure declarations is that CDD record declarations are language independent and can be used with several supported VAX languages.

The following examples demonstrate how data definitions are written for the CDD. The first example is a structure declaration written in CDDL. The second example shows the same structure as it would appear in a VAX FORTRAN output listing.

- CDDL Representation:

```
PAYROLL_RECORD STRUCTURE.
  SALESMAN STRUCTURE.
    NAME                 DATATYPE IS TEXT 30.
    ADDRESS              DATATYPE IS TEXT 40.
    SALESMAN_ID          DATATYPE IS UNSIGNED NUMERIC 5.
  END SALESMAN STRUCTURE.
END PAYROLL_RECORD STRUCTURE.
```

- VAX FORTRAN Source Code Representation:

```
STRUCTURE /PAYROLL_RECORD/
    STRUCTURE SALESMAN
        CHARACTER*30 NAME
        CHARACTER*40 ADDRESS
        STRUCTURE SALESMAN_ID
            CHARACTER*3 %FILL
        END STRUCTURE
    END STRUCTURE
END STRUCTURE
```

The CDD provides two utilities for creating and maintaining a dictionary:

- The Dictionary Management Utility (DMU)—used for creating and maintaining the CDD's directory hierarchy, history lists, and access control lists.
- The Dictionary Verify/Fix Utility (CDDV)—used for repairing damaged dictionary files.

See the *VAX Common Data Dictionary Utilities Reference Manual* for details.

1.4.1 Accessing the CDD from VAX FORTRAN Programs

DMU commands create directories and define record paths. Once these paths are established, records can be extracted from the CDD by means of DICTIONARY statements in VAX FORTRAN programs.

At compile time, the CDD record and its attributes are extracted from the designated CDD record node. Then, the compiler converts the extracted record into a VAX FORTRAN structure declaration and includes it in the object module.

The DICTIONARY statement incorporates VAX Common Data Dictionary data definitions into the current VAX FORTRAN source file during compilation. It can occur anywhere in a VAX FORTRAN source file that a specification statement (such as a STRUCTURE...END STRUCTURE block) is allowed. The format of the DICTIONARY statement is described in the *VAX FORTRAN Language Reference Manual.*

A DICTIONARY statement must appear as a statement by itself; it cannot be used within a VAX FORTRAN structure declaration. For example, consider the following DICTIONARY statement:

```
INTEGER*4 PRICE
DICTIONARY 'ACCOUNTS'
```

This would result in a declaration of the following form:

```
INTEGER*4 PRICE
STRUCTURE /ACCOUNTS/
    STRUCTURE NUMBER
        CHARACTER*3 LEDGER
        CHARACTER*5 SUBACCOUNT
    END STRUCTURE
    CHARACTER*12 DATE
    .
    .
    .
END STRUCTURE
```

When you extract a record definition from the CDD, you can choose to include this translated record in the program's listing by using the /LIST qualifier in the DICTIONARY statement or the /SHOW=DICTIONARY qualifier in the FORTRAN command line.

CDD data definitions can contain explanatory text in CDDL's DESCRIPTION IS clause. If you specify /SHOW=DICTIONARY on the FORTRAN command (or /LIST in the DICTIONARY statement), this text is included in the VAX FORTRAN output listing as comments.

Because the DICTIONARY statement generally contains only structure declaration blocks, you will usually also need to include one or more RECORD statements in your program to make use of these structures. (See the *VAX FORTRAN Language Reference Manual* for information about structure declaration blocks or the RECORD statement.)

1.4.2 Creating CDD Structure Declarations

CDD source files must be written in the Common Data Dictionary Language (CDDL). You enter them using a VMS editor (for example, EVE), just as you would any other file. After you have created a CDD source file, you can then invoke the CDD compiler to insert your record definitions into the CDD. See the *VAX CDD Data Definition Language Reference Manual* for detailed information about the CDDL language and compiler.

1.4.3 VAX FORTRAN and CDDL Data Types

The CDD supports some data types that are not native to VAX FORTRAN. If a data definition contains a field declared with an unsupported data type, VAX FORTRAN replaces the field with one declared as an inner STRUCTURE containing a single CHARACTER %FILL field of an appropriate length. The VAX FORTRAN compiler does not attempt to approximate a data type that it does not support. For example, CDD's data type UNSIGNED LONG is not supported by VAX FORTRAN. As a result, if the field FIELD1 is declared to be UNSIGNED LONG using CDDL, VAX FORTRAN would replace the definition of FIELD1 with the following declaration:

```
STRUCTURE FIELD1
    CHARACTER*4 %FILL
END STRUCTURE
```

VAX FORTRAN does not declare it as INTEGER*4, which would result in signed operations if the field was used in an arithmetic expression.

The following table summarizes the CDDL data types and corresponding VAX FORTRAN data types. For further information on CDDL data types see the *Common Data Dictionary Data Definition Language Reference Manual.*

CDDL Data Type	VAX FORTRAN Data Type
DATE	STRUCTURE (length 8)
DATE AND TIME	STRUCTURE (length n)
VIRTUAL	ignored
BIT m ALIGNED	STRUCTURE (length n+7/8)
BIT m	STRUCTURE (length n+7/8)
UNSPECIFIED	STRUCTURE (length n)
TEXT	CHARACTER*n
VARYING TEXT	STRUCTURE (length n)
VARYING STRING	STRUCTURE (length n)
D_FLOATING	REAL*8 (/NOG_FLOAT only)
D_FLOATING COMPLEX	COMPLEX*16 (/NOG_FLOAT only)
F_FLOATING	REAL*4
F_FLOATING COMPLEX	COMPLEX*8
G_FLOATING	REAL*8 (/G_FLOAT only)
G_FLOATING COMPLEX	COMPLEX*16 (/G_FLOAT only)
H_FLOATING	REAL*16
H_FLOATING COMPLEX	STRUCTURE (length 32)
SIGNED BYTE	LOGICAL*1
UNSIGNED BYTE	STRUCTURE (length 1)
SIGNED WORD	INTEGER*2
UNSIGNED WORD	STRUCTURE (length 2)
SIGNED LONGWORD	INTEGER*4
UNSIGNED LONGWORD	STRUCTURE (length 4)
SIGNED QUADWORD	STRUCTURE (length 8)
UNSIGNED QUADWORD	STRUCTURE (length 8)
SIGNED OCTAWORD	STRUCTURE (length 16)
UNSIGNED OCTAWORD	STRUCTURE (length 16)
PACKED NUMERIC	STRUCTURE (length n)

CDDL Data Type	VAX FORTRAN Data Type
SIGNED NUMERIC	STRUCTURE (length n)
UNSIGNED NUMERIC	STRUCTURE (length n)
LEFT OVERPUNCHED	STRUCTURE (length n)
LEFT SEPARATE	STRUCTURE (length n)
RIGHT OVERPUNCHED	STRUCTURE (length n)
RIGHT SEPARATE	STRUCTURE (length n)

NOTE

D_floating and G_floating data types cannot be mixed in one subroutine because both types cannot be handled simultaneously. You can use both types, each in a separate subroutine, depending on the OPTIONS statement qualifier in effect for the individual subroutine. See the *VAX FORTRAN Language Reference Manual* for a discussion of the handling of REAL*8 data types in VAX FORTRAN.

The compiler issues an error message whenever it encounters a CDD feature that conflicts with VAX FORTRAN. It ignores any CDD features that it does not support.

1.5 Compiler Diagnostic Messages and Error Conditions

One of the functions of the VAX FORTRAN compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to your default output device; thus, if you enter the FORTRAN command interactively, the messages are displayed on your terminal. If the FORTRAN command is executed in a batch job, the messages appear in the log file for the batch job.

When it appears on the terminal, a message from the compiler has the following form:

```
%FORT-s-ident, message-text
               [text-in-error] in module module-name at line n
```

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

Each compilation with diagnostic messages terminates with a summary that indicates the combined number of error, warning, and informational messages generated by the compiler. The diagnostic summary has the following form:

```
%FORT-s-ident, source-file-spec completed with n diagnostics
```

If the compiler creates a listing file, it also writes the messages to the listing. Messages typically follow the statement that caused the error.

Additional information about diagnostic messages, including descriptions of the individual messages, is contained in Appendix F.

1.6 Compiler Output Listing Format

A compiler output listing produced by a FORTRAN command with the /LIST qualifier consists of the following sections:

- A source code section
- A machine code section—optional
- A storage map section (cross-reference)—optional
- A compilation summary

Sections 1.6.1 through 1.6.4 describe the compiler listing sections in detail.

1.6.1 Source Code Section

The source code section of a compiler output listing displays the source program as it appears in the input file, with the addition of sequential line numbers generated by the compiler. Example 1–1 shows a sample of a source code section of a compiler output listing.

Compiler-generated line numbers appear in the left margin and are used with the %LINE prefix in debugger commands. If you create the source file with an editor that generates line numbers, those numbers 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

Example 1–1: Sample Listing of Source Code

```
0001                SUBROUTINE RELAX2(EPS)
0002
0003                PARAMETER (M=40, N=60)
0004                DIMENSION X(0:M,0:N)
0005                COMMON X
0006
0007                LOGICAL DONE
0008
0009      1         DONE = .TRUE.
0010
0011                DO 10 J=1,N-1
0012                DO 10 I=1,M-1
0013                   XNEW = (X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))/4
0014                   IF (ABS(XNEW-X(I,J)) .GT. EPS) DONE = .FALSE.
0015      10           X(I,J) = XNEW
0016
0017                IF (.NOT. DONE) GO TO 1
0018
0019                RETURN
0020                END
```

the right. The %LINE specification still applies to the compiler-generated line numbers, not the editor-generated line numbers.

If editor-generated line numbers are present in the source code listing, those numbers are reflected in compile-time error messages that contain line numbers. Otherwise, if editor-generated line numbers are not present in the source code listing, compiler-generated line numbers are used for this purpose. Run-time error messages that contain line numbers refer to the compiler-generated line numbers in the source code section of the compiler output listing. (See Appendix E for a summary of error messages.)

1.6.2 Machine Code Section

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

The machine code section is optional. To receive a listing file with a machine code section, you must specify the following:

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

Example 1–2 shows a sample of a machine code section of a compiler output listing.

Example 1–2: Sample Listing of Machine Code

```
                .TITLE  RELAX2
                .IDENT  01

0000            .PSECT  $BLANK
0000   X:

0000            .PSECT  $CODE
                                                          ; 0001
0000   RELAX2::
0000            .WORD   ^M<IV,R2,R3,R4,R6,R7>
                                                          ; 0009
0002            NOP
0003            NOP
0004   .1:
0004            MNEGL   #1, R0
                                                          ; 0011
0007            MOVL    #1, R1
000A            NOP
000B            NOP
000C   L$1:
                                                          ; 0012
000C            MOVL    #1, R2
000F            MULL3   #41, R1, R3
0013            MOVAF   X[R3], R4
001B            NOP
001C   L$2:
                                                          ; 0013
001C            ADDF3   8(R4), (R4)+, R6
0021            ADDF2   -164(R4), R6
0026            ADDF2   164(R4), R6
002B            MULF2   #^X3F80, R6
                                                          ; 0014
0032            SUBF3   (R4), R6, R7
0036            BICW2   #^X8000, R7
003B            CMPF    R7, @EPS(AP)
003F            BLEQ    L$3
0041            CLRL    R0
```

Example 1–2 Cont'd. on next page

Example 1–2 (Cont.): Sample Listing of Machine Code

```
0043  L$3:
                                                           ; 0015
0043        MOVL    R6, (R4)
0046        AOBLEQ  #39, R2, L$2
004A        AOBLEQ  #59, R1, L$1
                                                           ; 0017
004E        BLBC    R0, .1
                                                           ; 0019
0051        RET
            .END
```

The following list provides a detailed explanation of how generated code and data are represented in machine code listings.

- Machine instructions are represented by VAX MACRO mnemonics and syntax. To enable you to identify the machine code that is generated from a particular line of source code, the compiler-generated line numbers that appear in the source code listing are also used in the machine code listing. These numbers appear in the right margin, preceding the machine code generated from individual lines of source code.
- The first line contains a .TITLE assembler directive, indicating the program unit from which the machine code was generated.
  - For a main program, the title is as declared in a PROGRAM statement. If you did not specify a PROGRAM statement, the main program is titled filename$MAIN, where filename is the name of the source file.
  - For a subprogram, the title is the name of the subroutine or function.
  - For a BLOCK DATA subprogram, the title is either the name declared in the BLOCK DATA statement or filename$DATA (by default).
- The lines following .TITLE provide information such as the contents of storage initialized for FORMAT statements, DATA statements, constants, and subprogram argument call lists.
- The VAX 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 as they were defined in the program. Offsets from variables and arrays are shown in decimal.
- VAX FORTRAN source labels referenced in the source program are shown with a period 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 are not referenced or are deleted during compiler optimization, are ignored. They 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$n, where the value of n is unique for each such label in a program unit.
- Integer constants are shown as signed integer values; real 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.

1.6.3 Storage Map Section

The storage map section of the compiler output listing is printed after each program unit, or module. It is not generated when a fatal compilation error is encountered.

The storage map section summarizes information in the following categories:

- *Program sections*: 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 in bytes
  - Attributes

  PSECT usage and attributes are described in Section 10.1.

- *Total memory allocated*: Following the program sections, the compiler prints the total memory allocated for all program sections compiled in the following form:

```
Total Space Allocated nnn
```

- *Entry points*: 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.
- *Statement functions*: The statement function summary lists the entry point address and data type of each statement function. If all of the references to a statement function generate inline code, the body of the statement function is not compiled, and a double asterisk (**) appears instead of an address.
- *Variables*: The variable summary lists all simple variables, with the data type and address of each. If the variable is removed as a result of optimization, a double asterisk (**) appears in place of the address.
- *Records*: The record summary lists all record variables. It shows the address, the structure that defines the fields of the individual records, and the total size of each record.
- *Arrays*: The array summary is similar to the variable summary. In addition to data type and address, the array summary gives the total size and dimensions of the array. If the array is an adjustable array or assumed-size array, its size is shown as a double asterisk (**), and each adjustable dimension bound is shown as a single asterisk (*).
- *Record Arrays*: The record array summary is similar to the record summary. The record array summary gives the dimensions of the record array in addition to address, defining structure, and total size. If the record array is an adjustable array or assumed-size array, its size is shown as a double asterisk (**), and each adjustable dimension bound is shown as a single asterisk (*).
- *Namelists*: The namelist summary lists names of namelists.
- *Labels*: The label summary lists all user-defined statement labels. FORMAT statement labels are suffixed with an apostrophe ('). If the label address field contains a double asterisk (**), the label was not used or referred to by the compiled code.
- *Functions and subroutines*: The functions and subroutines summary lists all external routine references made by the source program. This summary does not include references to routines that are dummy arguments because the actual function or subroutine name is supplied by the calling program.

A heading for an information category is printed in the listing only when entries are generated for that category.

Cross-reference information is optional. It is supplied only when you specify the /LIST and /CROSS_REFERENCE qualifiers on the FORTRAN command line.

When you request cross-referencing, the compiler supplies information on the following entities:

- *Parameter constants*: The parameter constant summary lists all of the PARAMETER constants along with the data type of each.
- *Field scalars*: The field scalar summary lists all of the scalar fields declared within a structure block. It shows the starting offset within the structure for each scalar field, the name of the structure containing each scalar field, and the datatype and size (in bytes) of each scalar field.
- *Field arrays*: The field array summary lists all of the array fields declared within a structure block. It shows the starting offset within the structure for each array field; the name of the structure containing each array field; and the datatype, size (in bytes), and dimensions of each array field.

The compiler also supplies attributes and line-number references if you request cross-referencing, The attributes indicate whether a variable or array appears in common and whether it appears in an EQUIVALENCE statement.

The compiler supplies the following reference information for each name:

- A source line number indicates where the name was referenced.
- An equal sign (=) next to a line number indicates that the value of a variable or array was modified at that line.
- A number sign (#) next to a line number indicates the line where the symbol was defined.
- The character A next to a line number indicates an actual argument that may have been modified.
- The character D next to a line number indicates that data initialization occurred at that point in the program.
- A number in parentheses (n) next to a line number indicates that the name appeared n times on that line.

Example 1–3: Sample Storage Map Section

```
PROGRAM SECTIONS

    Name                                 Bytes   Attributes

  0 $CODE                                   82   PIC CON REL LCL   SHR   EXE   RD NOWRT LONG
  3 $BLANK                               10004   PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG

    Total Space Allocated                10086

ENTRY POINTS

    Address   Type  Name                     References

  0-00000000        RELAX2                       1#

VARIABLES

    Address   Type  Name          Attributes References

       **      L*4  DONE                         7         9=        14=        17
  AP-00000004@ R*4  EPS                          1         14
       **      I*4  I                            12=       13(4)     14         15
       **      I*4  J                            11=       13(4)     14         15
       **      R*4  XNEW                         13=       14        15


ARRAYS

    Address   Type  Name          Attributes       Bytes  Dimensions       References

  3-00000000  R*4   X             COMM             10004  (0:40, 0:60)     4        5        13(4)      14       15=

PARAMETER CONSTANTS

  Type  Name                                     References

   I*4  M                                        3#        4         12
   I*4  N                                        3#        4         11

LABELS

    Address    Label                             References

  0-00000004   1                                 9#        17
       **      10                                11        12        15#
```

Example 1–3 shows an example of a storage map section with cross-reference information.

As shown in Example 1–3, a section size is specified as a number of bytes, expressed in decimal. A data address is specified as an offset from the start of a program section, expressed in hexadecimal. The symbol AP can appear instead of a program section. When it does, 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.

1.6.4 Compilation Summary Section

The final entries on the compiler output listing are the compiler qualifiers and compiler statistics.

If the /CROSS_REFERENCE qualifier is specified, an explanation of the reference flags is printed at the top of the compilation summary.

The body of the compilation summary contains information about OPTIONS statement qualifiers (if any), FORTRAN command line qualifiers, and compilation statistics.

The first line under "Command Qualifiers" echoes the command line that you used to invoke the compiler. The set of qualifiers after the command line shows the qualifier defaults that were in effect during the compilation.

"Compiler Statistics" shows the machine resources used by the compiler.

Example 1–4 shows a sample compilation summary.

Example 1–4: Sample Compilation Summary

```
+---------------------------------------------+
|          KEY TO REFERENCE FLAGS             |
|  =   - Value Modified                       |
|  #   - Defining Reference                   |
|  A   - Actual Argument, possibly modified   |
|  D   - Data Initialization                  |
| (n)  - Number of occurrences on line        |
+---------------------------------------------+

OPTIONS QUALIFIERS

  /CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
  /F77 /NOG_FLOATING /NOI4


COMMAND QUALIFIERS

  FORTRAN /LISTING/MACHINE_CODE/CROSS_REFERENCE RELAX2

  /CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /SHOW=(NODICTIONARY,NOINCLUDE,MAP,NOPREPROCESSOR,SINGLE)
  /STANDARD=(NOSEMANTIC,NOSOURCE_FORM,NOSYNTAX)
  /WARNINGS=(NODECLARATIONS,GENERAL,NOULTRIX,NOVAXELN)
  /CONTINUATIONS=19 /CROSS_REFERENCE /NOD_LINES /NOEXTEND_SOURCE
  /F77 /NOG_FLOATING /I4 /MACHINE_CODE /OPTIMIZE /NOPARALLEL
  /NOANALYSIS_DATA
  /NODIAGNOSTICS
  /LIST=file-spec.LIS
  /OBJECT=file-spec.OBJ

COMPILATION STATISTICS

  Run Time:           1.17 seconds
  Elapsed Time:       2.23 seconds
  Page Faults:        138
  Dynamic Memory:     326 pages
```

Chapter 2

Linking and Running VAX FORTRAN Programs

This chapter describes how to produce an executable image from a VAX FORTRAN object file, how to execute the resulting image, and how to isolate run-time errors.

2.1 Linking VAX FORTRAN Programs

This section describes how to use the VMS Linker Utility and object module libraries to combine object modules into executable programs. It discusses the following topics:

- The functions performed by the linker
- The LINK command and its input and output files

The topics in this chapter are confined to areas of particular interest to VAX FORTRAN programmers. For additional information on linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VMS Linker Utility Manual*.

2.1.1 Functions of the Linker

The primary functions of the linker are to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation. The linker's end product is an executable image that you can run on a VMS system.

For any VAX FORTRAN program unit, the object module generated by the compiler may contain calls and references to VAX FORTRAN run-time procedures, which the linker locates automatically in the default system object module libraries. The libraries are described in the *VMS Linker Utility Manual.*

2.1.2 The LINK Command

The LINK command initiates the linking of the object file. The command has the following form:

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

/command-qualifiers
Specifies output file options.

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

/file-qualifiers
Specifies input file options.

In interactive mode, you can issue the LINK command without a file specification. The system then requests the file specifications with the following prompt:

```
_File:
```

You can enter multiple file specifications by separating them with commas (,) or plus signs (+). When used with the LINK command, the comma has the same effect as the plus sign; that is, 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 that of the first object module and with a file type of EXE. Table 2–1 lists the linker qualifiers of particular interest to VAX FORTRAN users.

Table 2–1: LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output file and define a file specification	/EXECUTABLE[=file-spec] /SHAREABLE[=file-spec]	/EXECUTABLE=name.EXE where *name* is the name of the first input file. /NOSHAREABLE
Request and specify the contents of a memory allocation listing.	/BRIEF /[NO]CROSS_REFERENCE /FULL /[NO]MAP	/NOCROSS_REFERENCE /NOMAP (interactive) /MAP=name.MAP (batch) where *name* is the name of the first input file.
Specify the amount of debugging information.	/[NO]DEBUG /[NO]TRACEBACK	/NODEBUG /TRACEBACK
Indicate that input files are libraries and specifically include certain modules.	/INCLUDE=(modulename...) /LIBRARY /SELECTIVE_SEARCH	Not applicable.
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB /[NO]SYSSHR /[NO]USERLIBRARY[=table]	/SYSLIB /SYSSHR /USERLIBRARY=ALL
Indicate that an input file is a linker options file.	/OPTIONS	Not applicable.

2.1.2.1 Linker Output File Qualifiers

You can use qualifiers on the LINK command line to control the output produced by the linker. You can also specify whether the debugging or the traceback facility is to be included. (The /DEBUG and /TRACEBACK qualifiers are described in Section 2.1.2.2.)

Linker output consists of an image file and, optionally, a map file. The qualifiers that control image and map files are described under the headings that follow.

Image File Qualifiers

The image file qualifiers are /[NO]EXECUTABLE and /[NO]SHAREABLE. The use and effects of these two qualifiers are as follows:

- **/EXECUTABLE**—If you do not specify an image file qualifier, the default is /EXECUTABLE, and the linker produces an executable image.

  To suppress production of an image, specify /NOEXECUTABLE. For example, in the following command, the file CIRCLE.OBJ is linked, but no image is generated:

  ```
  $ LINK/NOEXECUTABLE CIRCLE
  ```

  The /NOEXECUTABLE 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 /EXECUTABLE qualifier in the form:

  ```
     /EXECUTABLE=file-spec
  ```

  For example, in the following command, the file CIRCLE.OBJ is linked and the executable image generated by the linker is named TEST.EXE:

  ```
  $ LINK/EXECUTABLE=TEST CIRCLE
  ```

- **/SHAREABLE**—A shareable image has all of its internal references resolved, but must be linked with one or more object modules to produce an executable image. A shareable image file, for example, can contain a library of routines or can be used by the system manager to create a global section for all users. To create a shareable image, specify the /SHAREABLE qualifier, as shown in the following example:

  ```
  $ LINK/SHAREABLE CIRCLE
  ```

  To include a shareable image as input to the linker, you can insert the shareable image into a shareable-image library and specify the library as input to the LINK command. By default, the linker automatically searches the system-supplied shareable-image library SYS$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command line. You can also include a shareable image by using a linker options file. See the *VMS Linker Utility Manual* for more information.

If you specify (or default to) /NOSHAREABLE, the image produced cannot be linked with other images.

Map File Qualifiers

The map file qualifiers indicate whether a map file is to be generated and, if so, the amount of information to be included in the map file.

The map qualifiers are specified as follows:

```
/[NO]MAP[=file-spec] [ { /BRIEF } ] [/CROSS_REFERENCE]
                     [ { /FULL  } ]
```

In interactive mode, the default is to suppress the map; in batch mode, the default is to generate the map.

If you do not include a file specification with the /MAP qualifier, 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 /BRIEF and /FULL qualifiers define the amount of information included in the map file. They function 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). It also produces a list, in symbol-name order, of global symbols and values (program, subroutine, and common block names, and names declared EXTERNAL) and a summary of characteristics of image sections in the linked image.

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

You can use the /CROSS_REFERENCE qualifier 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 or refer to global symbols encountered during linking. The default is /NOCROSS_REFERENCE.

2.1.2.2 /DEBUG and /TRACEBACK Qualifiers

The /DEBUG qualifier indicates that the debugger (see Chapter 3) is to be included in the executable image and that local symbol information contained in the object modules is to be included. The default is /NODEBUG.

When you use the /TRACEBACK qualifier, run-time error messages are accompanied by a symbolic traceback that shows 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. (See Section 2.3.1 for a sample traceback list.)

2.1.2.3 Linker Input File Qualifiers

Input file qualifiers affect the file specifications of input files. Input files can be object files, shareable files previously linked, or library files.

The qualifiers that control linker input files are the /LIBRARY qualifier and the /INCLUDE qualifier.

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

  The /LIBRARY qualifier has the following form:

  ```
  /LIBRARY
  ```

- The /INCLUDE qualifier specifies that the input file is an object-module or shareable-image library and that the modules named are the only modules in the library to be explicitly included as input. In the case of shareable-image libraries, the module is the shareable-image name.

  The /INCLUDE qualifier has the following form:

  ```
  /INCLUDE=module-name
  ```

  At least one module name is required. To specify more than one, enclose the module names in parentheses and separate the names 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 to be searched for unresolved references.

2.1.3 Linker Messages

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors or fatal conditions occur (severities E or F), the linker does not produce an image file.

Linker messages are descriptive; you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking are as follows:

- An object module has compilation errors. This error occurs when you attempt to link a module that had warnings or errors during compilation. Although you can usually link compiled modules for which the compiler generated messages, you should verify that the modules actually produce the output you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one main program has been defined. This can occur, for example, when an extra END statement exists in the program. In this case, the image file created by the linker can be run; the entry point to which control is transferred is the first one that the linker finds.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names on the LINK command line and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link modules, you can often correct it simply by reentering the command string and specifying the correct routines or libraries.

2.2 Running VAX FORTRAN Programs

This section describes the following considerations for executing VAX FORTRAN programs on a VMS operating system:

- Using the RUN command to execute programs interactively
- Passing status values to the command interpreter

2.2.1 The RUN Command

The RUN command initiates execution of a program. The command has the following form:

```
RUN[/[NO]DEBUG] file-spec
```

You must specify the file name. If you omit optional elements of the file specification, the system automatically provides a default value. The default file type is EXE.

The /DEBUG qualifier allows you to use the debugger, even if you omitted this qualifier on the FORTRAN and LINK command lines. Refer to Section 2.3 for details.

Before the image is activated, the system initializes to zero all variables and arrays that are not initialized by means of DATA statements. (Note: It is not considered a good programming practice to rely on this, however.)

2.2.2 System Processing at Image Exit

When the main program executes an END statement, or when any program unit in the program executes a STOP statement, the image is terminated. In a VMS operating system, the termination of an image, or image exit, causes the system to perform a variety of clean-up operations during which open files are closed, system resources are freed, and so on.

2.2.3 Interrupting a Program

When you execute the RUN command interactively, you cannot execute any other program images or DCL commands until the current image completes. However, if your program is not performing as expected—if, for instance, you have reason to believe it is in an endless loop—you can interrupt it using the CTRL/Y key sequence. (You can also use the CTRL/C key sequence, unless your program takes specific action in response to CTRL/C.) For example:

```
$ RUN APPLIC
[CTRL/Y]
$
```

This command interrupts the program APPLIC. After you have interrupted a program, you can terminate it by entering a DCL command that causes another image to be executed or by entering the DCL commands EXIT or STOP.

Following a CTRL/Y interruption, you can also force an entry to the debugger by entering the DEBUG command.

Some of the other DCL commands you can enter have no direct effect on the image. After using them, you can resume the execution of the image with the DCL command CONTINUE. For example:

```
$ RUN APPLIC
[CTRL/Y]
$ SHOW TRANSLATION INFILE
  INFILE =    (undefined)
$ DEFINE INFILE DBA1:[TESTFILES]JANUARY.DAT
$ CONTINUE
```

For a complete list of the commands you can enter following a CTRL/Y interruption without affecting the current image, see the *VMS Command Definition Utility Manual.*

As noted previously, you can use CTRL/C to interrupt your program; in most cases, the effect of CTRL/C and CTRL/Y is the same. However, some programs (including programs you may write) establish particular actions to take to respond to CTRL/C. If a program has no CTRL/C handling routine, then CTRL/C is the same as CTRL/Y.

2.2.4 Returning Status Values to the Command Interpreter

If you run your program as part of a command procedure, it is frequently useful to return a status value to the command procedure indicating whether the program actually executed properly. To return such a status value, call the EXIT system subroutine rather than terminating execution with a STOP, RETURN, or END statement. The EXIT subroutine can be called from any executable program unit. It terminates your program and returns the value of the argument as the return status value of the program. See the *VAX FORTRAN Language Reference Manual* for a description of the EXIT subroutine.

When the command interpreter receives a status value from a terminating program, it attempts to locate a corresponding message in a system message file or a user-defined message file. Every message that can be issued by a system program, command, or component, has a unique 32-bit numeric value associated with it. These 32-bit numeric values are called condition symbols. Condition symbols are described in Section 9.1.2.3.

The command interpreter does not display messages on completion of a program under the following circumstances:

- The EXIT argument specifies the value 1, corresponding to SUCCESS.
- The program does not return a value. If the program terminates with a RETURN, STOP, or END statement, a value of 1 is always returned and no message is displayed.

2.3 Finding and Correcting Run-Time Errors

Both the compiler and the VMS Run-Time Library include facilities for detecting and reporting errors. You can use the VMS Debugger and the traceback facility to help you locate errors that occur during program execution.

2.3.1 Effects of 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 specify the /DEBUG qualifier on the FORTRAN command line to ensure that symbolic information is created for use by the debugger.
- At link time, you can also specify the /DEBUG qualifier on the LINK command line to make the symbolic information available to the debugger.
- At run time, you can specify the /DEBUG qualifier on the RUN command line to invoke the debugger.

Table 2-2 summarizes the /DEBUG and /TRACEBACK qualifiers.

Table 2-2: /DEBUG and /TRACEBACK Qualifiers

Command	Qualifier	Effect
FORTRAN	/DEBUG	The VAX FORTRAN compiler creates symbolic data needed by the debugger. **Default**: /DEBUG=(NOSYMBOLS,TRACEBACK)
LINK	/DEBUG	Symbolic data created by the VAX FORTRAN compiler is passed to the debugger. **Default**: /NODEBUG
	/TRACEBACK	Traceback information is passed to the debugger. Traceback will be produced. **Default**: /TRACEBACK
RUN	/DEBUG	Invokes the debugger. The DBG> prompt will be displayed. Not needed if $ LINK/DEBUG was specified.
	/NODEBUG	If /DEBUG was specified in the LINK command line, RUN/NODEBUG starts program execution without first invoking the debugger.

If an exception occurs and these qualifiers are not specified at any point in the compile-link-execute sequence, a traceback list is generated by default.

To perform symbolic debugging, you must use the /DEBUG qualifier with both the FORTRAN and LINK command lines; you do not need to specify it with the RUN command. If /DEBUG is omitted from either the FORTRAN or LINK command lines, you can still use it with the RUN command to invoke the debugger. However, any debugging you perform must then be done by specifying virtual addresses rather than symbolic names.

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

```
RUN/NODEBUG program-name
```

If you specify LINK/NOTRACEBACK, you receive no traceback in the event of errors. A sample source program and a traceback are shown in Example 2–1.

When an error condition is detected, you receive the appropriate message, followed by the traceback information. The Run-Time Library displays a message 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 can 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 are the values listed under "routine name" and "line." The names under "routine name" show which 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). With 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 and you do not receive a traceback list if an error occurs. To display traceback information, you can use the debugger command SHOW CALLS.

You should specify the /NOOPTIMIZE qualifier on the FORTRAN command line whenever you use the debugger; see Section 1.2.3.17.

Example 2–1: Sample VAX FORTRAN Program and Traceback

```
0001          PROGRAM TRACE_TEST
0002          I = 1
0003
0004          CALL SUB1(I)
0005          END

0001          SUBROUTINE SUB1(I)
0002          I = I + 1
0003          CALL SUB2
0004          RETURN
0005          END

0001
0002          SUBROUTINE SUB2
0003          COMPLEX W
0004          COMPLEX Z
0005
0006          DATA W/(0.,0.)/
0007          Z = LOG(W)
0008          RETURN
0009          END

%MTH-F-INVARGMAT, invalid argument to math library
  user PC 000034D4
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name    routine name    line    relative PC      absolute PC
                                         00001368         00001368
                                         00002C51         00002C51
                                         000034D4         000034D4
SUB2           SUB2               7      00000011         00000439
SUB1           SUB1               3      0000000C         00000424
TRACE_TEST     TRACE_TEST         4      00000014         00000414
```

Chapter 3

Using the VMS Debugger

This chapter is an introduction to using the VMS Debugger with VAX FORTRAN programs. This chapter provides the following information:

- An overview of the debugger (Sections 3.1 and 3.2)
- Information to get you started using the debugger (Section 3.3)
- A sample terminal session that demonstrates using the debugger (Section 3.4)
- A list of the debugger commands by function (Section 3.5)

For complete reference information on the VMS Debugger, see the *VMS Debugger Manual*. Online HELP is available during debugging sessions.

3.1 Overview

A debugger is a tool that helps you locate run-time errors quickly. It is used with a program that has already been compiled and linked successfully, but does not run correctly. For example, the output may be obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively so you can locate the point at which the program stopped working correctly.

The VMS Debugger is a *symbolic* debugger, which means that you can refer to program locations by the symbols (names) you used for those locations in your program—the names of variables, subroutines, labels, and so on. You do not need to use virtual addresses to refer to memory locations.

The debugger recognizes the syntax, expressions, data typing, and other constructs of VAX FORTRAN, as well as the following other languages supported on VAX:

Ada
BASIC
BLISS
C
COBOL
DIBOL
MACRO-32
Pascal
PL/I
RPG II
SCAN

If your program is written in more than one language, you can change from one language to another during a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

By issuing debugger commands at your terminal, you can perform the following operations:

- Start, stop, and resume the program's execution
- Trace the execution path of the program
- Monitor selected locations, variables, or events
- Examine and modify the contents of variables, or force events to occur
- Test the effect of some program modifications without having to edit, recompile, and relink the program

Such techniques allow you to isolate an error in your code much more quickly than you could without the debugger.

Once you have found the error in the program, you can then edit the source code and compile, link, and run the corrected version.

This chapter describes how to debug programs that run in only one process. Additional techniques for debugging multiprocess programs are covered in Appendix A.

3.2 Features of the Debugger

The VMS Debugger provides the following features to help you debug your programs:

- **Online HELP**—Online HELP is always available during a debugging session and contains information on all of the debugger commands and also information on selected topics.
- **Source Code Display**—You can display lines of source code during a debugging session.
- **Screen Mode**—You can capture and display various kinds of information in scrollable windows, which can be moved around the screen and resized. Automatically updated source, instruction, and register displays are available. You can selectively direct debugger input, output, and diagnostic messages to displays.
- **Keypad Mode**—When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad (if you have a VT100, VT52, or LK201 keyboard).
- **Source Editing**—As you find errors during a debugging session, you can use the EDIT command to invoke any editor available on your system. (You first specify the editor you want with the SET EDITOR command.)
- **Command Procedures**—The debugger allows you to execute a command procedure to recreate a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session.
- **Symbol Definitions**—You can define your own symbols to represent lengthy commands, address expressions, or values.
- **Initialization Files**—You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. In addition, you may want to have special initialization files for debugging specific programs.
- **Log Files**—You can record the commands you issue during a debugging session and the debugger's responses to those commands in a log file. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

3.3 Getting Started with the Debugger

This section explains how to use the debugger with VAX FORTRAN programs. The section focuses on basic debugger functions, to get you started quickly. It also provides any debugger information that is specific to VAX FORTRAN. For more detailed information that is not specific to a particular language, see the *VMS Debugger Manual*.

3.3.1 Compiling and Linking a Program to Prepare for Debugging

Before you can use the debugger, you must compile and link your program. The following example shows how to compile and link a VAX FORTRAN program (consisting of a single compilation unit named INVENTORY.FOR) prior to using the debugger.

```
$ FORTRAN/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the FORTRAN command line causes the compiler to write the debug symbol records associated with INVENTORY.FOR into the object module INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY.FOR in debugger commands. (If your program has several compilation units, each of the program units that you want to debug must be compiled with the /DEBUG qualifier.)

Use the /NOOPTIMIZE qualifier when you compile a program in preparation for debugging. Otherwise, the object code is optimized (to reduce the size of the program and make it run faster), so that the contents of some program locations may be inconsistent with what you might expect from viewing the source code. (After debugging the program, recompile it without the /NOOPTIMIZE qualifier.) See Chapter 11 for a detailed description of the various optimizations performed by the compiler and how these affect debugging.

The /DEBUG qualifier on the LINK command line causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. This qualifier also causes the VMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify the other modules on the LINK command line.) For a description of the effects of specifying the /DEBUG qualifier on the FORTRAN, LINK, and RUN command lines, see Sections 1.2.3.5 and 2.3.

3.3.2 Starting and Terminating a Debugging Session

You can invoke the debugger in either the *default* or *multiprocess* configuration to debug programs that run in either one or several processes, respectively. The configuration depends on the current value of the logical name DBG$PROCESS. Thus, before invoking the debugger, issue the DCL command SHOW LOGICAL DBG$PROCESS to determine the current definition of DBG$PROCESS.

This chapter covers programs that run in only one process. For such programs, DBG$PROCESS either should be undefined, as in the following example, or should have the value DEFAULT:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

If DBG$PROCESS has the value MULTIPROCESS, enter the following command to debug programs that run in only one process (see Appendix A for details on multiprocess debugging):

```
$ DEFINE DBG$PROCESS DEFAULT
```

You can now invoke the debugger by issuing the DCL command RUN. The following messages then appear on your screen:

```
$ RUN INVENTORY

                     VAX DEBUG Version Version 5.0

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'INVENTORY'
DBG>
```

The "INITIAL" message indicates that the debugging session is initialized for a VAX FORTRAN program and that the name of the main program unit is INVENTORY. The DBG> prompt indicates that you can now type debugger commands. At this point, if you type the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input or an error occurs).

When you invoke the debugger for either a mixed-language program that includes an Ada package or a program compiled with the /CHECK=UNDERFLOW or /PARALLEL qualifier, the following message, instead of the one shown previously, appears:

```
$ RUN INVENTORY

                     VAX DEBUG Version Version 5.0

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'INVENTORY'
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```

The "NOTATMAIN" message indicates that execution is suspended before the start of the main program, so that you can execute initialization code under debugger control. Typing the GO command places you at the start of the main program. At that point, type the GO command again to start program execution. Execution continues until it is forced to pause or stop (for example, if the program prompts you for input or if an error occurs).

To end a debugging session and return to DCL level, type EXIT or press CTRL/Z:

```
DBG> EXIT
$
```

The following message indicates that your program has completed execution successfully:

```
%DEBUG-I--EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

If you want to continue debugging after seeing this message, type EXIT and start a new debugging session with the DCL command RUN. You could also restart execution from within the debugging session with a command such as GO %LINE 1. However, this may produce unexpected results if, for example, some variables are initialized differently from when you first invoked the debugger.

3.3.3 Aborting Program Execution or Debugger Commands

If your program loops during a debugging session so that the debugger prompt does not reappear, press CTRL/C. This interrupts program execution and returns you to the prompt. For example:

```
DBG> GO
    .
    .
    .
(infinite loop)
[CTRL/C]
Interrupt
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

Do not press CTRL/Y from within a debugging session. Pressing CTRL/Y aborts the session and returns you to the DCL prompt ($) rather than the debugger prompt.

You can also press CTRL/C to abort the execution of a debugger command. This is useful if a command takes a long time to complete. For example:

```
DBG> EXAMINE/BYTE 1000:101000
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0)
CTRL/C      ! Should have typed 1000:1010
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

If your program already has a CTRL/C AST service routine enabled, use the SET ABORT_KEY command to assign the debugger's abort function to another CTRL-key sequence. For example:

```
DBG> SET ABORT_KEY = CTRL_P
DBG> GO
     .
     .
     .
CTRL/P
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

Note, however, that many CTRL-key sequences have VMS predefined functions, and the SET ABORT_KEY command enables you to override such definitions within the debugging session (see the *VMS DCL Concepts Manual*). Some of the CTRL-key characters not used by the VMS operating system are G, K, N, and P.

3.3.4 Issuing Debugger Commands

You can issue debugger commands any time you see the debugger prompt (DBG>). Type the command at the keyboard and press the RETURN key. You can issue several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the previous line with a hyphen (-).

Alternatively, you can use the numeric keypad to issue certain commands. Figure 3–1 identifies the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE.

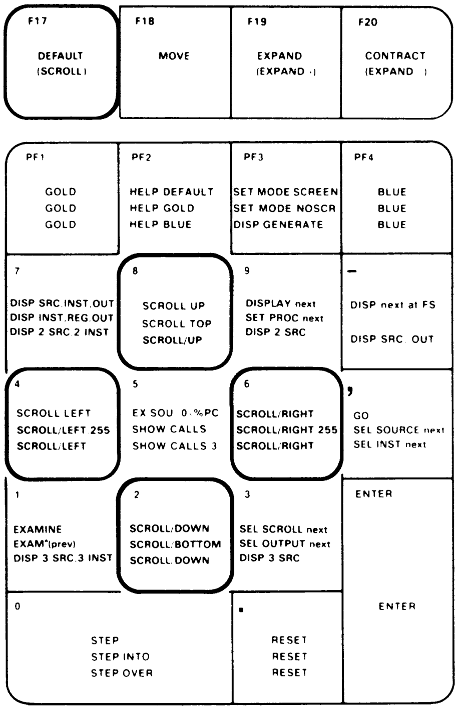
- To obtain a key's DEFAULT function, press the key.
- To obtain its GOLD function, first press the PF1 (GOLD) key, and then the key.
- To obtain its BLUE function, first press the PF4 (BLUE) key, and then the key.

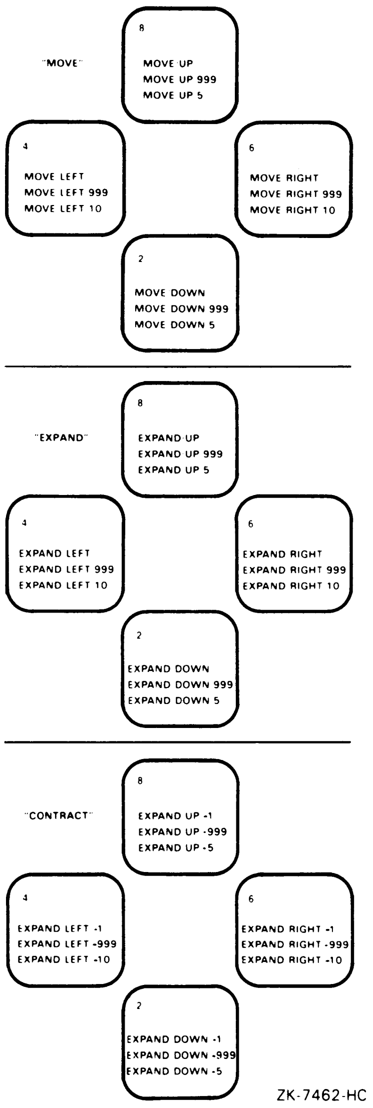
In Figure 3–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom, respectively. For example,

- Pressing keypad key 0 issues the STEP command.
- Pressing key PF1 and then key 0 issues the STEP/INTO command.
- Pressing key PF4 and then key 0 issues the STEP/OVER command.

Type the command HELP KEYPAD to get help on the keypad key definitions.

Figure 3–1: Debugger Keypad Key Functions





LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

ZK-7462-HC

3.3.5 Viewing Your Source Code

The debugger provides two modes for displaying information: noscreen mode and screen mode. By default, when you invoke the debugger, you are in noscreen mode, but you may find that it is easier to view your source code in screen mode. Both modes are briefly described in the following sections.

3.3.5.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To invoke noscreen mode from screen mode, press the keypad key sequence GOLD-PF3. See the sample debugging session in Section 3.4 for a demonstration of noscreen mode.

In noscreen mode, you can use the TYPE command to display one or more source lines. For example, the following command displays line 3 of the module whose code is currently executing:

```
DBG> TYPE 3
module MAIN
     3:    J = 4
DBG>
```

The display of source lines is independent of program execution. To display source code from a module other than the one whose code is currently executing, use the TYPE command with a path name to specify the module. (See the description of the STEP command in Section 3.3.6.1 for information about path names.)

For example, the following command displays lines 16 through 21 of module TEST:

```
DBG> TYPE TEST\16:21
```

You can also use the EXAMINE/SOURCE command to display the source line for a routine or any other program location that is associated with an instruction.

Note that the debugger also displays source lines automatically when it suspends execution at a breakpoint or watchpoint or after a STEP command, or when a tracepoint is triggered (see Section 3.3.6).

If the debugger cannot locate source lines for display, it issues a diagnostic message. Source lines may not be available for a variety of reasons. For example:

- The module was compiled or linked without the /DEBUG command qualifier.
- Execution is currently suspended within a system or shareable image routine for which no source code is available.
- The module may need to be set with the SET MODULE command. Module setting is explained in Section 3.3.8.1.
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules). In this case, use the SET SOURCE command to specify the new location.

3.3.5.2 Screen Mode

To invoke screen mode, press keypad key PF3. In screen mode, the debugger splits the screen into three displays named SRC, OUT, and PROMPT, by default. The following example shows how your screen will appear in screen mode.

```
- SRC: module MAIN -scroll-source-----------------------
     1:         PROGRAM MAIN
     2:         I = 7
->   3:         J = 4
     4:         K = I + J
     5:         END




- OUT -output-------------------------------------------
stepped to MAIN\%LINE 4
MAIN\I: 7
MAIN\J: 0


- PROMPT -error-program-prompt--------------------------
DBG> STEP 2
DBG> EXAMINE I,J
DBG>
```

The SRC display, at the top of the screen, shows the source code of the module (compilation unit) where execution is currently suspended. An arrow in the left column points to the next line to be executed, which corresponds to the current value of the program counter, PC. (The PC is a VAX register that contains the address of the next instruction to be executed.) The line numbers, which are assigned by the compiler, match those in a listing file.

The OUT display, in the middle of the screen, captures the debugger's output in response to the commands that you issue.

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output. In the example, the two debugger commands that have been issued (STEP 2 and EXAMINE I,J) are displayed.

(The zero value reported by the debugger for J indicates that line 3 has not been executed yet; line 3 will subsequently assign the value 4 to J.)

The SRC and OUT displays can be scrolled to display information beyond the window's edge. Press keypad key 8 to scroll up and keypad key 2 to scroll down. Use keypad key 3 to change the display to be scrolled (by default, the SRC display is scrolled). Scrolling a display does not affect program execution.

In screen mode, if the debugger cannot locate source lines for the program unit where execution is currently suspended, it tries to display source lines in the next routine down on the call stack for which source lines are available. If this is possible, the debugger also issues the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
        Displaying source in a caller of the current routine.
```

In such cases, the arrow in the SRC display identifies the call statement in the calling routine.

3.3.6 Controlling and Monitoring Program Execution

This section discusses the following topics:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining where execution is currently suspended with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

3.3.6.1 Starting and Resuming Program Execution—GO and STEP

The GO and STEP commands allow you to start or resume program execution. The GO command starts execution, and the STEP command executes a specified number of source lines or instructions.

The GO Command

The GO command is usually issued only after you have established breakpoints, tracepoints, and watchpoints (described in Sections 3.3.6.3, 3.3.6.4, and 3.3.6.5).

- If you set a breakpoint in the path of execution and then issue the GO command, execution is suspended at that breakpoint.
- If you set a tracepoint, the path of execution through that tracepoint is monitored.
- If you set a watchpoint, execution is suspended when the value of the watched variable changes.

You can also use the GO command to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger takes control and displays the DBG> prompt so that you can issue commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You can use the SHOW CALLS command (explained in Section 3.3.6.2) to identify the currently active routine calls (the call stack).

If an infinite loop occurs, the program does not terminate, so the debugger prompt does not reappear. To obtain the prompt, interrupt execution by pressing CTRL/C (see Section 3.3.3). You can then look at the source display and a display generated by the SHOW CALLS command to find where execution is suspended.

The STEP Command

The STEP command allows you to execute a specified number of source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next CALL instruction.

By default, the STEP command executes a single executable source line at a time. In the following example, the STEP command executes one line, reports the action ("stepped to . . . "), and displays the line number (27) and source code of the next line to be executed:

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
     27:   X = X + 1
DBG>
```

Execution is now suspended at the first machine code instruction for line 27 of the module TEST; line 27 is in COUNT, a subroutine within the module TEST. TEST\COUNT\%LINE 27 is a *path name*. The debugger uses path names to refer to symbols. (You do not need to use a path name in referring to a symbol, however, unless the symbol is not unique. If the symbol is not unique, the debugger issues an error message. See Section 3.3.8.2 for more information on resolving multiply defined symbols.)

The STEP command can execute a number of lines at a time. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
```

Note that only those source lines for which code instructions were generated by the compiler are recognized as executable lines by the debugger. The debugger skips over any other lines, for example, comment lines and specification statements.

You can specify different stepping modes, such as stepping by instruction rather than by line (SET STEP INSTRUCTION). You may also want to adjust the stepping mode used when a routine call is encountered in the source code. By default, the debugger steps over called routines; execution is not suspended within a called routine, although the routine is executed. Issuing the SET STEP INTO command causes the debugger to suspend execution within called routines, as well as within the routine that is currently executing.

3.3.6.2 Determining Where Execution Is Suspended—SHOW CALLS

The SHOW CALLS command is useful when you are unsure where execution is suspended during a debugging session (for example, after a CTRL/C interruption).

The SHOW CALLS command displays a traceback that lists the sequence of calls leading to the routine where execution is currently suspended. For each routine (beginning with the one where execution is suspended), the debugger displays the following information:

- The name of the module that contains the routine
- The name of the routine
- The line number at which the call was made (or at which execution is suspended, in the case of the current routine)
- The corresponding PC addresses (the relative PC address from the start of the routine, and the absolute PC address of the program)

For example:

```
DBG> SHOW CALLS
 module name        routine name         line      rel PC    abs PC

*TEST               PRODUCT                18    00000009  0000063C
*TEST               COUNT                  47    00000009  00000647
*MY_PROG            MY_PROG                21    0000000D  00000653
DBG>
```

This example indicates that execution is currently at line 18 of routine PRODUCT (in module TEST), which was called from line 47 of routine COUNT (in module TEST), which was called from line 21 of routine MY_PROG (in module MY_PROG).

3.3.6.3 Suspending Program Execution—SET BREAK

The SET BREAK command allows you to select *breakpoints*. Breakpoints are locations at which program execution is suspended. When you reach a breakpoint, you can issue commands to check the call stack, examine the current values of variables, and so on.

In the following example, the SET BREAK command sets a breakpoint on the subroutine COUNT. The GO command then starts execution. When the subroutine COUNT is encountered, execution is suspended. The debugger reports that the breakpoint at COUNT has been reached ("break at . . . "), displays the source line (54) where execution is suspended, and prompts you for another command. At this breakpoint, you could step through the subroutine COUNT, using the STEP command, and use the EXAMINE command (discussed in Section 3.3.7.1) to check on the current values of X and Y.

```
DBG> SET BREAK COUNT
DBG> GO
     .
     .
     .
break at PROG2\COUNT
     54:  SUBROUTINE COUNT(X,Y)
DBG>
```

When using the SET BREAK command, you can specify program locations using various kinds of *address expressions* (for example, line numbers, subroutine names, instructions, virtual memory addresses, or byte offsets). With high-level languages, you typically use subroutine names, labels, or line numbers, possibly with path names to ensure uniqueness.

Subroutine names and labels should be specified as they appear in the source code. Line numbers may be derived from either a source code display or a listing file. When specifying a line number, use the prefix %LINE, and when specifying a label, use the prefix %LABEL. (Otherwise, the debugger interprets the line numbers as memory locations.) For example, the next command sets a breakpoint at line 41 of the module whose code is currently executing; the debugger suspends execution when the PC value is at the start of line 41.

```
DBG> SET BREAK %LINE 41
```

Note that you can set breakpoints only on lines that resulted in machine code instructions. If you try to do otherwise (for example, if you try to set a breakpoint on a comment line), the debugger issues a warning. To set a breakpoint on a line number in a module other than the one whose code is currently executing, specify the module's name in a path name. For example:

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

You do not always need to specify a particular program location, such as line 58 or COUNT, to set a breakpoint. You can set breakpoints on events, such as exceptions. You can also use the SET BREAK command with the /LINE qualifier (but no parameter) to break on every line, or with the /CALL qualifier to break on every CALL instruction, and so on. For example:

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

You can conditionalize a breakpoint (with a WHEN clause) or specify that a list of commands be executed at the breakpoint (with a DO clause). For example, the next command sets a breakpoint on the label LOOP3. The DO (EXAMINE TEMP) clause causes the value of the variable TEMP to be displayed whenever the breakpoint is triggered.

```
DBG> SET BREAK %LABEL 10 DO (EXAMINE TEMP)
DBG> GO
         .
         .
         .
break at COUNTER\%LABEL 10
     37:     10    DO I = 1 TO 10
COUNTER\TEMP:   284.19
DBG>
```

To display the currently active breakpoints, issue the SHOW BREAK command:

```
DBG> SHOW BREAK
breakpoint at SCREEN_IO\%LINE 58
breakpoint at COUNTER\%LABEL 10
   do (EXAMINE TEMP)
         .
         .
         .
DBG>
```

If any portion of your program was written in Ada, two breakpoints that are associated with Ada tasking exception events are automatically established when you invoke the debugger. When you issue a SHOW BREAK command under these conditions, the following breakpoints are displayed:

```
DBG> SHOW BREAK
Breakpoint on ADA event "DEPENDENTS_EXCEPTION" for any value
Breakpoint on ADA event "EXCEPTION_TERMINATED" for any value
```

These breakpoints are equivalent to issuing the following commands:

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
```

To cancel a breakpoint, issue the CANCEL BREAK command, specifying the program location or event exactly as you did when setting the breakpoint. The CANCEL BREAK/ALL command cancels all breakpoints.

3.3.6.4 Tracing Program Execution—SET TRACE

The SET TRACE command allows you to select *tracepoints* Tracepoints are locations for tracing the execution of your program without stopping its execution. After setting a tracepoint, you can start execution with the GO command and then monitor the path of execution, checking for unexpected behavior. By setting a tracepoint on a routine, you can also monitor the number of times the routine is called.

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. However, at tracepoints, the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE COUNT
DBG> GO
     .
     .
     .
trace at PROG2\COUNT
     54:  SUBROUTINE COUNT(X,Y)
     .
     .
     .
```

When using the SET TRACE command, specify address expressions, qualifiers, and optional clauses exactly as with the SET BREAK command.

The /LINE qualifier causes the SET TRACE command to trace every line and is a convenient means of checking the execution path. By default, lines are traced within all called routines, as well as the currently executing routine. If you do not want to trace through system routines or through routines in shareable images, use the /NOSYSTEM or /NOSHARE qualifiers. For example:

```
DBG> SET TRACE/LINE/NOSYSTEM/NOSHARE
```

The /SILENT qualifier suppresses the trace message and the display of source code. This is useful when you want to use the SET TRACE command to execute a debugger command at the tracepoint. For example:

```
DBG> SET TRACE/SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
     .
     .
     .
SCREEN_IO\CLEAR\STATUS:   'OFF'
     .
     .
     .
```

3.3.6.5 Monitoring Changes in Variables—SET WATCH

The SET WATCH command allows you to set *watchpoints* that will be monitored continuously as your program executes. With high-level languages, you typically set watchpoints on variables (and, occasionally, on arbitrary program locations). If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

To set a watchpoint on a variable, specify the variable's name with the SET WATCH command. For example, the following command sets a watchpoint on the variable TOTAL:

```
DBG> SET WATCH TOTAL
```

Subsequently, every time the program modifies the value of TOTAL, the watchpoint is triggered.

The following example shows the effect on program execution when your program modifies the contents of a watched variable.

```
DBG> SET WATCH TOTAL
DBG> GO
     .
     .
     .
watch of SCREEN_IO\TOTAL at SCREEN_IO\%LINE 13
     13:   TOTAL = TOTAL + 1
    old value: 16
    new value: 17
break at SCREEN_IO.%LINE 14
     14:   CALL POP(TOTAL)
DBG>
```

In this example, a watchpoint is set on the variable TOTAL, and the GO command is issued to start execution. When the value of TOTAL changes, execution is suspended. The debugger reports the event ("watch of . . . ") and identifies where TOTAL changed (line 13) and the associated source line. The debugger then displays the old and new values and reports that execution has been suspended at the start of the next line (14). (The debugger reports "break at . . . ", but this is not a breakpoint; it is the effect of the watchpoint.) Finally, the debugger prompts for another command.

When a change in a variable occurs at a point other than at the start of a source line, the debugger gives the line number plus the byte offset from the start of the line.

Note that this general technique for setting watchpoints always applies to "static" variables. A static variable is associated with the same virtual memory location throughout program execution.

A variable that is allocated on the stack or in a register (a "nonstatic" variable) exists only when its defining routine is active (on the call stack). If you try to set a watchpoint on a nonstatic variable when its defining subroutine is not active, the debugger issues a warning:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'Y' is not active
```

A convenient technique for setting a watchpoint on a nonstatic variable is to set a breakpoint on the defining subroutine, also specifying a DO clause to set the watchpoint whenever execution reaches the breakpoint. In the following example, a watchpoint is set on the nonstatic variable Y in routine COUNTER:

```
DBG> SET BREAK COUNTER DO (SET WATCH Y)
DBG> GO
    .
    .
    .
break at routine MOD4\COUNTER
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG> SHOW WATCH
watchpoint of MOD4\COUNTER\Y [tracing every instruction]
DBG>
```

The debugger monitors nonstatic watchpoints by tracing every instruction. Because this slows execution speed compared to monitoring static watchpoints, the debugger lets you know when it is monitoring nonstatic watchpoints.

When execution eventually returns to the calling routine, the nonstatic variable is no longer active, so the debugger automatically cancels the watchpoint and issues a message to that effect.

As explained in Section 3.3.1, if you specify the /OPTIMIZE qualifier (or take the default) when compiling your program, certain variables in your program may be removed by the compiler. If you try to set a watchpoint on one of these variables, the debugger issues the following warning:

```
%DEBUG-W-UNALLOCATED, entity 'symbol' was not allocated in memory (was optimized away)
```

3.3.7 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables and to evaluate expressions. It also notes restrictions on the use of these commands with VAX FORTRAN programs.

Note that, before you can examine or deposit into a nonstatic variable (as defined in the previous section), its defining routine must be active (that is, currently residing on the call stack).

3.3.7.1 Displaying the Values of Variables—EXAMINE

To display the current value of a variable, use the EXAMINE command. The EXAMINE command has the following form:

```
  EXAMINE variable-name
```

The debugger recognizes the compiler-generated data type of the specified variable and retrieves and formats the data accordingly. The following examples show some uses of the EXAMINE command.

Examine a string variable:

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

Examine a two-dimensional array of integers (three per dimension):

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
   (1,1):       27
   (1,2):       31
   (1,3):       12
   (2,1):       15
   (2,2):       22
   (2,3):       18
DBG>
```

Examine element 4 of a one-dimensional array of characters:

```
DBG> EXAMINE CHAR_ARRAY(4)
PROG2\CHAR_ARRAY(4): 'M'
DBG>
```

The EXAMINE command can be used with any kind of address expression, not just a variable name, to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format.

See Section 3.3.7.3 for a comparison of the EXAMINE and EVALUATE commands.

3.3.7.2 Changing the Values of Variables—DEPOSIT

To change the value of a variable, use the DEPOSIT command. The DEPOSIT command has the following form:

```
   DEPOSIT variable-name = value
```

The DEPOSIT command is like an assignment statement in VAX FORTRAN.

In the following examples, the DEPOSIT command assigns new values to different variables. The debugger checks that the value assigned, which can be a language expression, is consistent with the data type and dimensional constraints of the variable.

Deposit a string value (it must be enclosed in quotation marks or apostrophes):

```
DBG> DEPOSIT PARTNUMBER = "WG-7619.3-84"
```

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH = CURRENTWIDTH + 10
```

Deposit element 12 of an array of characters:

```
DBG> DEPOSIT C_ARRAY(12) = 'K'
```

Note that you cannot deposit an entire array aggregate with a single DEPOSIT command, only an element.

As with the EXAMINE command, the DEPOSIT command lets you specify any kind of address expression, not just a variable name. You can override the defaults for typed and untyped locations if you want the data to be interpreted in some other data format.

3.3.7.3 Evaluating Expressions—EVALUATE

The EVALUATE command allows you to evaluate a language expression. The EVALUATE command has the following form:

```
   EVALUATE language-expression
```

The debugger recognizes the operators and expression syntax of the currently set language. In the following example, the value 45 is assigned to the integer variable WIDTH; the EVALUATE command then obtains the sum of the current value of WIDTH plus 7:

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

In the next example, the values .TRUE. and .FALSE. are assigned to the LOGICAL*1 variables WILLING and ABLE, respectively, and the EVALUATE command then obtains the logical conjunction of these values:

```
DBG> DEPOSIT WILLING = .TRUE.
DBG> DEPOSIT ABLE = .FALSE.
DBG> EVALUATE WILLING .AND. ABLE
0
DBG>
```

The following example shows how the EVALUATE and EXAMINE commands are similar. When the expression following the command is a variable name, the value reported by the debugger is the same for either command.

```
DBG> DEPOSIT WIDTH = 45
DBG> EVALUATE WIDTH
45
DBG> EXAMINE WIDTH
SIZE\WIDTH:   45
```

The following example shows an important difference between the EVALUATE and EXAMINE commands:

```
DBG> EVALUATE WIDTH + 7
52
DBG> EXAMINE WIDTH + 7
SIZE\WIDTH:   131584
```

With the EVALUATE command, WIDTH + 7 is interpreted as a language expression, which evaluates to 45 + 7, or 52. With the EXAMINE command, WIDTH + 7 is interpreted as an address expression: Seven bytes are added to the address of WIDTH, and whatever value is in the resulting address is reported (in this instance, 131584).

3.3.7.4 Notes on Debugger Support for VAX FORTRAN

In general, the debugger supports the data types and operators of VAX FORTRAN and of the other debugger-supported languages. However, there are certain language-specific limitations or other differences. (For information on the supported data types and operators of any of the languages, type the HELP LANGUAGE command at the DBG> prompt.)

- Even though the VAX type codes for unsigned integers (BU, WU, LU) are used internally to describe the LOGICAL data types, the debugger (like the compiler) treats LOGICAL variables and values as being signed when used in language expressions.
- The debugger prints the numeric values of LOGICAL variables or expressions instead of .TRUE. or .FALSE. Normally, only the low-order bit of a LOGICAL variable or value is significant (0 is .FALSE. and 1 is .TRUE.). However, VAX FORTRAN does allow all bits in a LOGICAL value to be manipulated and LOGICAL values can be used in integer expressions. For this reason, it is at times necessary to see the entire integer value of a LOGICAL variable or expression, and that is what the debugger shows.
- COMPLEX constants such as (1.0,2.0) are not supported in debugger expressions.
- Floating point numbers of type REAL*8 and COMPLEX*16 may be represented by D_Floating or G_Floating depending on compiler switches.

3.3.8 Controlling Symbol References

In most cases, the way the debugger handles symbols (variable names, and so on) is transparent to you. However, the following two areas may require action on your part:

- Module setting
- Multiply defined symbols

3.3.8.1 Module Setting—SET MODULE

To facilitate symbol searches, the debugger loads symbol records from the executable image into a run-time symbol table (RST), where they can be accessed efficiently. Unless a symbol record is in the RST, the debugger cannot recognize or properly interpret that symbol.

Because the RST uses memory, the debugger loads it dynamically, anticipating what symbols you might want to reference during execution. The loading process is called *module setting* because all of the symbol records of a given module are loaded into the RST at one time.

At debugger startup, only the module containing the image transfer address is set. As your program executes, whenever the debugger interrupts execution, it sets the module where execution is suspended. This allows you to reference the symbols that should be visible at that location.

If you try to reference a symbol in a module that has not been set, the debugger issues a warning. For example:

```
DBG> EXAMINE K
%DEBUG-W-NOSYMBOL, symbol 'K' is not in symbol table
DBG>
```

You must then use the SET MODULE command to manually set the module containing that symbol:

```
DBG> SET MODULE MOD3
DBG> EXAMINE K
MOD3\ROUT2\K: 26
DBG>
```

The SHOW MODULE command lists the modules of your program and identifies which modules have been set.

Note that dynamic module setting may slow down the debugger as more and more modules are set. If performance becomes a problem, you can reduce the problem with the CANCEL MODULE or SET MODE NODYNAMIC commands:

- Use the CANCEL MODULE command to reduce the number of modules that are set.
- Use the SET MODE NODYNAMIC command to disable dynamic module setting. (The SET MODE DYNAMIC command enables dynamic module setting.)

3.3.8.2 Resolving Multiply Defined Symbols

The debugger finds the symbols that you reference in commands according to the following conventions. First, it looks in the PC scope (also known as *scope 0*), according to the scope and visibility rules of the currently set language. This means that the debugger first searches for a symbol within the routine surrounding the current PC value (where execution is currently suspended). If the symbol is not found, the debugger searches the nesting program unit, then its nesting unit, and so on. (The precise order of search depends on the currently set language and guarantees that the proper declaration of a multiply defined symbol is selected.)

The debugger allows you to reference symbols throughout your program, not just those that are visible at the current PC value. This enables you to set breakpoints in arbitrary areas, examine arbitrary variables, and so on. Therefore, if the symbol is not visible in the PC scope, the debugger also searches the scope of the calling routine (if any), then its caller, and so on, until the symbol is found. Symbolically, this search list is denoted 0,1,2, . . . ,*n*, where scope 0 is the PC scope and *n* is the number of calls in the call stack. Within each scope, the debugger uses the visibility rules of the currently set language to locate symbols.

If the debugger cannot resolve a symbol ambiguity, it issues a warning. For example:

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG>
```

You can then use a path-name prefix to uniquely specify a declaration of the given symbol. First, use the SHOW SYMBOL command to identify all path names associated with the given symbol; then use the desired path name when referencing the symbol. For example:

```
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
DBG> EXAMINE MOD4\ROUT2\Y
MOD4\ROUT2\Y: 12
DBG>
```

If you need to refer to a particular declaration of Y repeatedly, use the SET SCOPE command to establish a new default scope for symbol lookup. Then, references to Y without a path-name prefix will specify the declaration of Y that is visible in the new scope region. For example:

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

To display the current scope for symbol lookup, use the SHOW SCOPE command. To restore the default scope, use the CANCEL SCOPE command.

3.4 Sample Debugging Session

The sample debugging session presented in this section involves the following source program:

```
            INTEGER INARR(20), OUTARR(20)
    C
    C       ---Read the input array from the data file.
            OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
            READ(8,*) N, (INARR(I), I=1,N)
    C
    C       ---Square all nonzero elements and store in OUTARR.
            K = 0
            DO 10 I = 1, N
            IF(INARR(I) .NE. 0) THEN
               OUTARR(K) = INARR(I)**2
            ENDIF
    10      CONTINUE
    C
    C       ---Print the squared output values.  Then stop.
            PRINT 20, K
    20      FORMAT(' Number of nonzero elements is',I4)
            DO 40 I = 1, K
            PRINT 30, I, OUTARR(I)
    30      FORMAT(' Element',I4,' has value',I6)
    40      CONTINUE
            END
```

The program reads a sequence of integer numbers from a data file (lines 4 and 5) and saves these numbers in the array INARR. The program then enters a loop (lines 8 through 13) where it copies the square of each nonzero integer into another array, OUTARR. Finally, the program prints the number of nonzero elements in the original sequence and the square of each such element (lines 16 through 21).

The error in the program occurs when variable K, which keeps track of the current index into OUTARR, is not incremented in the loop on lines 9 through 13. The statement K = K + 1 should be inserted just before line 11.

To find this error, first compile, link, and run the program as follows:

```
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES
$ LINK/DEBUG SQUARES
$ RUN SQUARES

         VAX DEBUG Version Version 5.0

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'SQUARES$MAIN'
DBG>
```

You can now issue debugger commands. To step forward four lines, type the following command:

```
DBG> STEP 4
stepped to SQUARES$MAIN\%LINE 9
DBG>
```

To check the current values of variables N and K, type the following command:

```
DBG> EXAM N, K
SQUARES$MAIN\N:       9
SQUARES$MAIN\K:       0
DBG>
```

The values of N and K are both correct at this point in the execution. Now, issue the command STEP 2 to enter the loop that copies and squares all nonzero elements of INARR into OUTARR.

```
DBG> STEP 2
stepped to SQUARES$MAIN\%LINE 11
DBG>
```

To check whether I and K have the expected values, type the following command:

```
DBG> EXAM I,K
SQUARES$MAIN\I:       1
SQUARES$MAIN\K:       0
DBG>
```

I has the expected value (namely 1), but K has the value 0, which is not the expected value. Now you can see the error in the program: K should be incremented in the loop just before it is used in line 11. To check this hypothesis, patch the program by issuing the following debugger commands:

```
DBG> DEPOSIT K = 1
DBG> SET TRACE/SILENT %LINE 11 DO(DEPOSIT K = K + 1)
DBG>
```

The first command gives K the value it should have now, namely 1. The second command specifies that the debugger should perform the debugger command DEPOSIT K = K + 1 each time line 11 is reached and just before it is executed. The /SILENT qualifier suppresses the "trace at" message that would otherwise appear each time line 11 is executed. The program is now patched and should perform correctly.

Line 22 is a suitable location to set a breakpoint that will stop program execution after testing the correctness of your patch. To set a breakpoint at that line, type the following command:

```
DBG> SET BREAK %LINE 22
DBG>
```

Now, run your program to test your patch. Type the GO command to execute the program until it reaches the breakpoint at line 22.

```
DBG> GO
Number of nonzero elements is   6
Element   1 has value    16
Element   2 has value    36
Element   3 has value     9
Element   4 has value    49
Element   5 has value    81
Element   6 has value     1

break at SQUARES$MAIN\%LINE 22
     22:        END
DBG>
```

The program output shows that the program appears to work properly with the DEPOSIT K = K + 1 patch. You can now use the EDIT command to invoke the VAX Language-Sensitive Editor, or another editor previously established with the SET EDITOR command:

```
DBG> EDIT
```

The editor positions the cursor at the same line that is marked by the pointer in the debugger's source display.

The corrected portion of the source code is as follows:

```
          .
          .
          .
 8:      K = 0
 9:      DO 10 I = 1, N
10:      IF(INARR(I) .NE. 0) THEN
11:          K = K + 1
12:          OUTARR(K) = INARR(I)**2
13:      ENDIF
14:   10 CONTINUE
          .
          .
          .
```

Now, you can compile, link, and run the program again under debugger control to check that it behaves correctly:

```
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES
$ LINK/DEBUG SQUARES
$ RUN SQUARES
```

To set a breakpoint at line 12 that will display the values of I and K automatically, type the following command (the subsequent GO command starts execution):

```
DBG> SET BREAK %LINE 12 DO (EXAMINE I,K)
DBG> GO
     .
     .
     .
SQUARES$MAIN\I:        1
SQUARES$MAIN\K:        1
DBG> GO
     .
     .
     .
SQUARES$MAIN\I:        2
SQUARES$MAIN\K:        2
DBG> GO
     .
     .
     .
SQUARES$MAIN\I:        4
SQUARES$MAIN\K:        3
```

At the first breakpoint, the value of K is 1, indicating that the program is executing correctly thus far. Each additional GO command shows the current values of I and K. After two GO commands, K is now 3, as expected, but note that I is 4. The reason is that one of the INARR elements was 0, so lines 11 and 12 were not executed (and K was not incremented) on one iteration of the DO loop. This confirms that the program is executing correctly.

3.5 Debugger Command Summary

This section lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

During a debugging session, you can get online HELP on any command and its qualifiers by typing the HELP command followed by the name of the command in question. The HELP command has the following form:

```
HELP command
```

3.5.1 Starting and Terminating a Debugging Session

(\$) RUN[1]	Invokes the debugger if LINK/DEBUG was used
(\$) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed
CTRL/Z or EXIT	Ends a debugging session, executing all exit handlers
QUIT	Ends a debugging session without executing any exit handlers declared in the program
CTRL/C	Aborts program execution or a debugger command without interrupting the debugging session
$\left\{ \begin{array}{l} \text{SET} \\ \text{SHOW} \end{array} \right\}$ ABORT_KEY	Assigns the default CTRL/C abort function to another CTRL-key sequence or identifies the CTRL-key sequence currently defined for the abort function
(\$) CTRL/Y - DEBUG[1]	The sequence CTRL/Y - DEBUG interrupts a program that is running without debugger control and invokes the debugger
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH)
SPAWN	Creates a subprocess; lets you issue DCL commands without interrupting your debugging context (similar to the DCL command SPAWN)

[1]This is a DCL command, not a debugger command.

3.5.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
$\left\{ \begin{array}{l} \text{SET} \\ \text{SHOW} \end{array} \right\}$ STEP	Establishes or displays the default qualifiers for the STEP command

$\left\{\begin{array}{l}\text{SET}\\ \text{SHOW}\\ \text{CANCEL}\end{array}\right\}$ BREAK	Sets, displays, or cancels breakpoints
$\left\{\begin{array}{l}\text{SET}\\ \text{SHOW}\\ \text{CANCEL}\end{array}\right\}$ TRACE	Sets, displays, or cancels tracepoints
$\left\{\begin{array}{l}\text{SET}\\ \text{SHOW}\\ \text{CANCEL}\end{array}\right\}$ WATCH	Sets, displays, or cancels watchpoints
SHOW CALLS	Identifies the currently active subroutine calls
SHOW STACK	Gives additional information about the currently active subroutine calls
CALL	Calls a subroutine

3.5.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location
SET MODE [NO]OPERANDS	Controls whether the address and contents of the instruction operands are displayed when you examine an instruction
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression

3.5.4 Controlling Type Selection and Symbolization

$\left\{\begin{array}{l}\text{SET}\\ \text{SHOW}\\ \text{CANCEL}\end{array}\right\}$ RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix
$\left\{\begin{array}{l}\text{SET}\\ \text{SHOW}\\ \text{CANCEL}\end{array}\right\}$ TYPE	Establishes the type for program locations that are not associated with a compiler generated type, displays the type, or restores the type
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT

3.5.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image
SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are set automatically when the debugger interrupts execution
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ SCOPE	Establishes, displays, or restores the scope for symbol lookup
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

3.5.6 Displaying Source Code

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ SOURCE	Creates, displays, or cancels a source directory search list
SEARCH	Searches the source code for the specified string

{SET SHOW} SEARCH	Establishes or displays the default qualifiers for the SEARCH command
SET STEP [NO]SOURCE	Enables or disables the display of source code after a STEP command has been executed or at a breakpoint, tracepoint, or watchpoint
{SET SHOW} MAX_SOURCE_FILES	Establishes or displays the maximum number of source files that may be kept open at one time
{SET SHOW} MARGINS	Establishes or displays the left and right margin settings for displaying source code

3.5.7 Using Screen Mode

SET MODE [NO]SCREEN	Enables or disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
{SET SHOW CANCEL} DISPLAY	Creates, identifies, or deletes a display
{SET SHOW CANCEL} WINDOW	Creates, identifies, or deletes a window definition
SELECT	Selects a display for a display attribute
SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display and writes it to another display
EXTRACT	Saves a display or the current screen state and writes it to a file
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen

{SET / SHOW} TERMINAL	Establishes or displays the height and width of the screen
{CTRL/W / DISPLAY/REFRESH}	Refreshes the screen

3.5.8 Editing Source Code

EDIT	Invokes an editor during a debugging session
{SET / SHOW} EDITOR	Establishes or identifies the editor invoked by the EDIT command

3.5.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, value, or process group
DELETE	Deletes symbol definitions
{SET / SHOW} DEFINE	Establishes or displays the default qualifier for the DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined

3.5.10 Using Keypad Mode

SET MODE [NO]KEYPAD	Enables or disables keypad mode
DEFINE/KEY	Creates key definitions
DELETE/KEY	Deletes key definitions
SET KEY	Establishes the key definition state
SHOW KEY	Displays key definitions

3.5.11 Using Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures
$\left\{ \begin{array}{l} \text{SET} \\ \text{SHOW} \end{array} \right\}$ LOG	Specifies or identifies the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command
$\left\{ \begin{array}{l} \text{SET} \\ \text{SHOW} \end{array} \right\}$ ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures
@file-spec	Executes a command procedure

3.5.12 Using Control Structures

IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively
WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop

3.5.13 Debugging Multiprocess Programs

CONNECT	Brings a process under debugger control
DEFINE/PROCESS_GROUP	Assigns a symbolic name to a list of process specifications

DO	Executes commands in the context of one or more processes
SET MODE [NO]INTERRUPT	Controls whether execution is interrupted in other processes when it is suspended in some process
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ PROCESS	Modifies the multiprocess debugging environment or displays process information
SET PROMPT /[NO]SUFFIX	Enables you to specify a process-specific prompt-string suffix
DISPLAY, SET DISPLAY /[NO]PROCESS /SUFFIX	/PROCESS makes an existing display process specific or creates a process specific display. /SUFFIX appends a process identifying suffix to a display name (may be used with any command that specifies a display).

3.5.14 Additional Commands

SET PROMPT	Specifies the debugger prompt
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ LANGUAGE	Establishes or displays the current language
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ EVENT_FACILITY	Establishes or identifies the current run-time facility for language-specific events
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ TASK	Modifies the tasking environment or displays task information
$\left\{\begin{array}{l}\text{DISABLE}\\\text{ENABLE}\\\text{SHOW}\end{array}\right\}$ AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled
SET MODE [NO]SEPARATE	Controls whether a separate window is created on a MicroVAX workstation for debugger input and output (this command has no effect on VT-series terminals)

Chapter 4
VAX FORTRAN Input/Output

This chapter describes FORTRAN input/output (I/O) as implemented for VAX FORTRAN and provides information about FORTRAN I/O in relation to the VAX Record Management Services (RMS) and the VAX Run-Time Library (RTL).

The following topics are addressed in this chapter:

- Overview of FORTRAN I/O (Section 4.1)
- Elements of I/O Processing (Section 4.2)

4.1 Overview of VAX FORTRAN I/O

This section introduces the concept of logical units, briefly describes the scope of interprocess communications, and lists and describes the different types of I/O statements and the optional forms of I/O statements.

4.1.1 Identifying Logical Input/Output Units

Logical unit numbers are integers from 0 to 99. For example:

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

This READ 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.

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

READ, WRITE, and REWRITE statements refer explicitly to a logical unit from which or to which data is to be transferred. The logical unit can be connected to a device or file by means of an OPEN statement. (See the *VAX FORTRAN Language Reference Manual* for more information on the OPEN statement.)

ACCEPT, TYPE, and PRINT statements do not refer explicitly to a logical unit (a file or device) from which or to which data is to be transferred; they refer implicitly to a default logical unit. The ACCEPT statement is normally connected to the default input device, and the TYPE and PRINT statements are normally connected to the default output device. These defaults can be overridden with appropriate logical name assignments (see Section 4.2.2.1).

4.1.2 Types of I/O Statements

The type of an I/O statement depends on the organization of the file being accessed. The various types of I/O are as follows:

- Sequential I/O—transfers records sequentially to or from files or I/O devices such as terminals.
- Direct Access I/O—transfers records selected by record number to and from sequential (fixed length) or relative organization files.
- Keyed I/O—transfers records, based on data values (keys) contained in the records, to and from indexed files.
- Internal I/O—transfers data between variables and arrays defined within a program.

4.1.3 Interprocess Communication

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 permit interprocess communication on different computers. DECnet can also be used to access files on different computers.

Information on the preceding types of operations is provided in Chapter 8.

4.1.4 Forms of I/O Statements

Each type of I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. The I/O statement forms are *formatted, unformatted, list-directed,* and *namelist-directed.*

- Formatted I/O statements contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.
- List-directed and namelist-directed I/O statements are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist-directed I/O statements use data types.
- Unformatted I/O statements do not contain format specifiers and therefore do not translate the data being transferred. Unformatted I/O is especially appropriate where the output data will subsequently be used as input data. Unformatted I/O saves execution time by eliminating the data translation process, preserves greater precision in the external data, and usually conserves file storage space.

I/O statements transfer all data as records. The amount of data that a record can contain depends on whether you use unformatted or formatted I/O to transfer the data. With unformatted I/O, the I/O statement alone specifies the amount of data to be transferred; with formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Normally, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for formatted, list-directed, and namelist-directed I/O statements to transfer data from or to more than one record.

Table 4–1 shows the various I/O statements, by category, that can be used in FORTRAN programs.

Table 4–1: Available I/O Statements

Statement Category	Statement Name					
	READ	WRITE	REWRITE	ACCEPT	TYPE	PRINT
Sequential						
Formatted	Yes	Yes	No	Yes	Yes	Yes
List-Directed	Yes	Yes	No	Yes	Yes	Yes
Namelist-Directed	Yes	Yes	No	Yes	Yes	Yes
Unformatted	Yes	Yes	No	No	No	No
Direct						
Formatted	Yes	Yes	Yes	No	No	No
Unformatted	Yes	Yes	No	No	No	No
Indexed						
Formatted	Yes	Yes	Yes	No	No	No
Unformatted	Yes	Yes	Yes	No	No	No
Internal						
Formatted	Yes	Yes	No	No	No	No
List-Directed	Yes	Yes	No	No	No	No
Unformatted	No	No	No	No	No	No

4.2 Elements of I/O Processing

This section describes, in general terms, the following elements of VAX FORTRAN I/O processing:

- VMS file specifications (Section 4.2.1)
- Logical names, as used in FORTRAN, and logical unit numbers (Section 4.2.2)

- FORTRAN file organizations, I/O record formats, and access modes (Section 4.2.3)

4.2.1 File Specifications

The information about file specifications is abbreviated in this section, concentrating on how to identify files in I/O statements. For a detailed description of VMS file specifications, see the *Guide to VMS File Applications.*

A complete VMS file specification has the form:

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

For example:

```
BOSTON::USERD:[SMITH]TEST.DAT;2
```

You can associate a file specification with a logical unit by using a logical name assignment (see Section 4.2.2) or by using the OPEN statement (see Section 4.2.2.3). If you do not specify such an association or 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.
- 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, if your default device is USERD and your default directory is SMITH, and you specified the following statements:

```
READ (8,100)
   .
   .
   .
WRITE (9,200)
```

Then, the default input and output file specifications, respectively, would be as follows:

```
   USERD:[SMITH]FOR008.DAT;n
```

and

```
   USERD:[SMITH]FOR009.DAT;m
```

Where n equals the highest current version number of FOR008.DAT and m is 1 greater than the highest existing version number of FOR009.DAT.

4.2.2 Logical Names and Logical Unit Numbers

You can use the logical name mechanism of the VMS operating system to associate logical units with file specifications. A logical name is an alphanumeric string, up to 63 characters long, that you can use instead of a file specification.

The operating system supplies a number of predefined logical names that are already associated with particular file specifications. Table 4–2 lists the logical names of special interest to FORTRAN users. FORTRAN logical unit names are shown in Table 4–3.

Table 4–2: Predefined System Logical Names

Name	Meaning	Default
SYS$COMMAND	Default command stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job input command file.
SYS$DISK	Default disk device	As specified by user.
SYS$ERROR	Default error stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch job log file.
SYS$INPUT	Default input stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch command file.
SYS$OUTPUT	Default output stream	For an interactive user, the default is the terminal; for a batch job, the default is the batch log file.

You can dynamically create a logical name and associate it with a file specification by means of the VMS commands ASSIGN or DEFINE. For example, before program execution, you can associate the logical names in your program with the file specification appropriate to your needs.

For example:

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

The preceding command creates the logical name LOGNAM and associates it with the file specification USERD:[SMITH]TEST.DAT;2. As a result, this file specification is used whenever the logical name LOGNAM is encountered during program execution.

Logical names provide great flexibility because they can be associated with either a partial or complete file specification (that is, with either a device or a device and a directory), or even another logical name.

4.2.2.1 FORTRAN Logical Names

Usually, VAX FORTRAN I/O is performed by associating a logical unit number with a device or file. VMS logical names provide an additional level of association; a user-specified logical name can be associated with a logical unit number.

VAX FORTRAN provides predefined logical names in the following form:

```
FOROnn
```

The notation nn represents a logical unit number.

By default, each FORTRAN logical name is associated with a file named FOR0nn.DAT on your default disk under your default directory. For example:

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

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

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

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

The preceding command associates the FORTRAN logical name FOR017 (and therefore logical unit 17) with file TEST.DAT;2 on device USERD in directory [SMITH].

You can also associate the FORTRAN logical names with any of the predefined system logical names, as shown in the following examples:

- The following command associates the default command stream with the default input device (for example, the batch input stream):

  ```
  $ ASSIGN SYS$INPUT SYS$COMMAND
  ```

- The following command associates logical unit 10 with the default output device (for example, the batch output stream):

  ```
  $ ASSIGN SYS$OUTPUT FOR010
  ```

4.2.2.2 Implied FORTRAN Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and optionally the READ and WRITE statements use implicit logical unit numbers and names. They do not include explicit logical unit numbers, and each logical name is, in turn, associated by default with one of the system's predefined logical names. Table 4–3 shows these relationships.

Table 4–3: Implicit FORTRAN Logical Units

	FORTRAN	System
READ (*,f) iolist	FOR$READ	SYS$INPUT
READ f,iolist	FOR$READ	SYS$INPUT
ACCEPT f,iolist	FOR$ACCEPT	SYS$INPUT
WRITE (*,f) iolist	FOR$PRINT	SYS$OUTPUT
PRINT f,iolist	FOR$PRINT	SYS$OUTPUT
TYPE f,iolist	FOR$TYPE	SYS$OUTPUT
READ (5),iolist	FOR005	SYS$INPUT
WRITE (6),iolist	FOR006	SYS$OUTPUT

You can change the file specifications associated with these FORTRAN logical names, as you would any other FORTRAN logical name, by means of the DCL commands ASSIGN or DEFINE. For example:

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

Following execution of the preceding command, the READ statement's logical name (FOR$READ) refers to the file TEST.DAT;2 on device USERD in directory [SMITH].

4.2.2.3 File Specification in the OPEN Statement

You can use the FILE and DEFAULTFILE keywords of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (The *VAX FORTRAN Language Reference Manual* describes the OPEN statement in greater detail.) For example:

```
OPEN (UNIT=4, FILE='USERD:[SMITH]TEST.DAT;2', STATUS='OLD')
```

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

In the following interactive example, the file name is supplied by the user and the DEFAULTFILE keyword supplies the default values for the file specification string.

```
CHARACTER*9 DOC
TYPE *, 'ENTER FILE NAME (WITHIN APOSTROPHES)'
ACCEPT *, DOC
OPEN (UNIT=2, FILE=DOC,
1     DEFAULTFILE='USERD:[ARCHIVE].TXT',
1     STATUS='OLD')
```

In the preceding example, the file to be opened is located on device USERD in directory ARCHIVE, with the file name supplied in DOC and the file type TXT. The DEFAULTFILE specification overrides your process default device and directory.

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

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

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

```
OPEN (UNIT=19, FILE='LOGNAM', STATUS='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 FILE 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 USERD:[SMITH]TEST.DAT by means of an ASSIGN or DEFINE command, then the above OPEN statement indicates that a file named LOGNAM.DAT is located on the default device, in the default directory.

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

4.2.2.4 Assigning Files to Logical Units—Summary

As described in the preceding sections, you can assign files to logical units in the following ways:

- By using default logical names. In the following example, the READ statement causes the logical unit FOR007 to be associated with the file FOR007.DAT by default, and the TYPE statement causes the logical unit FOR$TYPE to be associated with SYS$OUTPUT by default.

  ```
  READ (7,100)
      .
      .
      .
  TYPE 100
  ```

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

  ```
  OPEN (UNIT=7, FILE='LOGNAM', STATUS='OLD')
  ```

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

  ```
  OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
  ```

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

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

Use the DCL command ASSIGN or DEFINE to change the association of logical names and file specifications.

Use the DCL command DEASSIGN to remove the association of a logical name and a file specification. For example:

```
$ DEASSIGN logical-name
```

4.2.3 File Organizations, I/O Record Formats, and Access Modes

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

- The term *file organization* applies to the way records are physically arranged on a storage device.
- The term *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. In contrast, record access is specified each time the file is opened, and can be different each time.

The following sections describe in general terms the elements of FORTRAN I/O processing: file organizations, internal files, I/O record formats, and record access modes.

4.2.3.1 File Organizations

A file is a collection of logically related records that are arranged in a specific order and treated as a unit. The arrangement, or organization, of a file is determined when the file is created.

VAX FORTRAN supports three kinds of file organization: sequential, relative, and indexed. The organization of a file is specified by means of the ORGANIZATION keyword in the OPEN statement, as described in the section on the OPEN statement in the *VAX FORTRAN Language Reference Manual.*

Files are normally stored on disk. Sequential files, however, can be stored on either magnetic tape or disk. Other peripheral devices, such as terminals, card readers, and line printers, are treated as sequential files.

The three kinds of file organization are discussed individually under the headings that follow.

Sequential Organization

A sequentially organized file consists 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, and so on). 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 VMS operating system.

Relative Organization

A relative file consists of numbered positions, called cells. These cells are of fixed equal length and are consecutively numbered from 1 to n, where 1 is the first cell, and n is the last available cell in the file. Each cell either contains a single record or is empty.

Records in a relative file are accessed according to cell number. A cell number is a record's relative record number, that is, its location relative to the beginning of the file. By specifying relative record numbers, you can directly retrieve, add, or delete records—regardless of their locations.

Relative files are supported only on disk devices.

Indexed Organization

An indexed file consists of two or more separate sections: one section contains the data records and the other sections contain indexes. When an indexed file is created, each index is associated with a specification defining a field, called a key field, within each record. A record in an indexed file must contain at least one key. This mandatory key, called the primary key, determines the location of the records within the body of the file.

The keys of all records are collected to form one or more structured indexes, through which records are always accessed. The structure of the indexes allows a program to access records in an indexed file either randomly, by specifying particular key values, or sequentially, by retrieving records with increasing key values. In addition, keyed access and sequential access can be mixed. The term Indexed Sequential Access Method (ISAM) refers to this dynamic access feature.

Indexed files are supported only on disk devices. See Chapter 14 for more information on indexed files.

4.2.3.2 Internal Files

An internal file is designated internal storage space that is manipulated to facilitate internal I/O. Its use with formatted and list-directed sequential READ and WRITE statements eliminates the need to use the ENCODE and DECODE statements for internal I/O.

An internal file consists of a character variable, a character array element, a character array, or a character substring; a record in an internal file consists of any of these data items except a character array.

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the variable, array element, or substring. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined; that is, a value has been assigned to the record.

Prior to data transfer, an internal file is always positioned at the beginning of the first record.

4.2.3.3 I/O Record Formats

An I/O record is a collection of data items, called fields, that are logically related and are processed as a unit.

NOTE

> I/O records are not to be confused with record entities declared in a program as structured data items. There is no relationship between structured data items and I/O records. Structured data items are described in the *VAX FORTRAN Language Reference Manual.*

Generally, each FORTRAN I/O statement transfers one record. The exceptions are formatted, list-directed, and namelist-directed I/O statements, which can transfer additional records.

If an input statement does not use all of the data fields in a record, the remaining fields are ignored. If an input statement requires more data fields than the record contains, either an error condition occurs or, in the case of formatted input, all fields are read as spaces.

If an output statement attempts to write more data fields than the record can contain, an error condition occurs. If an output statement transfers less data than is required to fill a fixed-length record, the record is filled with spaces (if it is a formatted record) or zeros (if it is an unformatted record).

Records are stored in one of four formats:

- Fixed-length
- Variable-length
- Segmented
- Stream

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

The various kinds of I/O record formats are discussed individually under the headings that follow.

Fixed-Length Records

When you specify fixed-length records, 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. (The *VAX FORTRAN Language Reference Manual* discusses fixed-length records.) A sequentially organized file opened for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

Variable-Length Records

Variable-length records can contain any number of bytes, up to a specified maximum. These 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 RECL keyword of the OPEN statement (see the *VAX FORTRAN Language Reference Manual* for details). This RECL value specifies the largest record that can be stored in the file.

The count field of a variable-length record is available when you read the record by issuing 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.

Segmented Records

A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequentially organized file. 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 but cannot or do not wish to define one long variable-length record. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

Because there is no 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
- The last segment
- The only segment
- None of the above

This control information is contained in the first two bytes of each segment of a segmented record. Therefore, when you wish to access an unformatted sequential file that contains variable-length records, you must specify RECORDTYPE='VARIABLE' when you open the file. Otherwise, the first two bytes of each record will be mistakenly interpreted as control information, and errors will probably result.

Stream Records

A stream-type record is a variable-length record whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

There are three varieties of stream-type files, each using a different record terminator:

- STREAM files use the 2-character sequence consisting of a carriage-return and a line-feed as the record terminator.
- STREAM_CR files use only a carriage-return as the terminator.
- STREAM_LF files use only a line-feed as the terminator.

4.2.3.4 Record Access Modes

Access mode is the method a program uses to retrieve and store records in a file. The access mode is specified as part of each I/O statement. VAX FORTRAN supports three record access modes:

- Sequential
- Direct
- Keyed

Your choice of record access mode is affected by the organization of the file to be accessed. For example, the sequential access mode can be used with sequential, relative, and indexed files; but the keyed access mode can be used only with indexed organization files.

Table 4–4 shows all of the valid combinations of access mode and file organization.

Table 4–4: Valid Combinations of Record Access Mode and File Organization

File Organization	Access Mode		
	Sequential	**Direct**	**Keyed**
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

[1]Fixed-length records only.

The three kinds of access mode are discussed individually under the headings that follow.

Sequential Access Mode

If you select sequential access mode for files with sequential or relative organization, records are written to or read from the file starting at the beginning and continuing through the file, one record after another. For files with indexed organization, sequential access can be used to read or write all records according to the direction of the key and the key values. Sequential access to indexed files can also be used with keyed access to read or write a group of records at a specified point in the file.

When you use sequential access for files with sequential and relative organization, a particular record can be retrieved only after all of the records preceding it have been read.

Writing records by means of sequential access also varies according to the file organization.

- For a file with sequential organization, new records can be written only at the end of the file.
- For a file with relative organization, a new record can be written at any point, replacing the existing record in the specified cell. For example, if two records are read from a relative file and then a record is written, the new record occupies cell 3 of the file.
- For a file with indexed organization, records can be written in any order, and READ operations refer to the next record with the same or next higher specified key value.

Direct Access Mode

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. You can also access a sequential disk file directly if it contains fixed-length records. 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, the following statements, appearing in a program in the order shown here, read record 24 and then read record 10.

```
READ (12,REC=24) I
READ (12,REC=10) J
```

Keyed Access Mode

If you select keyed access mode, you determine the order in which records are read or written by means of character values or integer values called keys. Each READ statement contains the key that locates the record. The key value in the I/O statement is compared with index entries until the record is located.

When you insert a new record, the values contained in the key fields of the record determine the record's placement in the file; you do not have to indicate a key.

You can use keyed access only for files with an indexed organization.

Your program can mix keyed access and sequential access I/O statements on the same file. You can use keyed I/O statements to position the file to a particular record and then use sequential I/O statements to access records with increasing key values in the current key-of-reference.

Chapter 5
Error Processing

During execution, your program may encounter errors or exception conditions. These conditions can result from errors that occur during I/O operations, from invalid input data, from argument errors in calls to the mathematical library, from arithmetic errors, or from system-detected errors. The Run-Time Library provides default processing for error conditions, generates appropriate messages, and takes action to recover from errors whenever possible. You can, however, explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the error (ERR) and end-of-file (END) specifiers in I/O statements.
- To identify FORTRAN-specific errors based on the value of IOSTAT, use the I/O status specifier (IOSTAT) in I/O statements.
- To tailor error processing to the special requirements of your applications, use the VAX condition-handling facility (including user-written condition handlers). (See Chapter 9 for information on user-written condition handlers.)

These error-processing methods are complementary; you can use all of them within the same program. However, before attempting to write a condition handler, you should be familiar with the VAX condition-handling facility (CHF) and with the condition-handling description in Chapter 9.

This chapter describes how the Run-Time Library processes errors. It also provides information about using I/O specifiers for explicit error processing and control, and shows how these methods affect the default error processing of the Run-Time Library.

5.1 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 error recognized by the Run-Time Library. The default actions described throughout this chapter occur unless overridden by explicit error-processing methods.

The way in which the Run-Time Library actually processes errors depends upon several factors:

- The severity of the error.
- Whether an I/O error-handling specifier or a condition handler was used.
- Whether the error permits continuation.

Table 5–1 lists the FORTRAN-specific errors processed by the Run-Time Library. For each error, the table shows the message text, the symbolic condition name, the FORTRAN-specific error code, and the severity code. (Refer to Table E-4 for more detailed descriptions of errors processed by the Run-Time Library.)

The condition symbols shown in the left column are the status codes signaled by the Run-Time Library I/O support routines. You can define these symbolic values in your program by including the module $FORDEF from the system-supplied default library FORSYSDEF.TLB.

The error numbers shown in the second column are the standard DIGITAL FORTRAN error numbers that are compatible with other versions of DIGITAL FORTRAN. These are the error values returned to IOSTAT variables when an I/O error is detected. These numbers are also used to index the error table maintained by the ERRSET and ERRTST subroutines. (See Appendix E for descriptions of the ERRSET and ERRTST subroutines.)

The codes in the third column indicate the severity of the error conditions. All FORTRAN-specific errors have severity codes of either error (E) or severe error (F). As shown in the table, most FORTRAN-specific errors are severe. If no explicit recovery action is specified for a severe error, program execution terminates by default.

The letter C in the severity column of the table means that you can continue execution immediately after the error, if a user-written condition handler specifies that execution should continue. If the letter C is not present, you cannot continue execution immediately after the error. If you attempt to do so, program execution terminates.

When errors occur for which no recovery method is specified, the program exits; that is, an error message is printed and execution of the program terminates. To prevent program termination, you must include an appropriate I/O error-handling specifier (see Sections 5.2 and 5.3) or a condition handler that performs an unwind. (See Chapter 9 for information on user-written condition handlers.)

Table 5–1: Summary of Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR$_NOTFORSPE[1]	1	F	Not a FORTRAN-specific error
FOR$_SYNERRNAM	17	F	Syntax error in NAMELIST input
FOR$_TOOMANVAL	18	F	Too many values for NAMELIST variable
FOR$_INVREFVAR	19	F	Invalid reference to variable in NAMELIST input
FOR$_REWERR	20	F	REWIND error
FOR$_DUPFILSPE	21	F	Duplicate file specifications
FOR$_INPRECTOO	22	F	Input record too long
FOR$_BACERR	23	F	BACKSPACE error
FOR$_ENDDURREA	24	F	End-of-file during read
FOR$_RECNUMOUT	25	F	Record number outside range
FOR$_OPEDEFREQ	26	F	OPEN or DEFINE FILE required
FOR$_TOOMANREC	27	F	Too many records in I/O statement
FOR$_CLOERR	28	F	CLOSE error
FOR$_FILNOTFOU	29	F	File not found
FOR$_OPEFAI	30	F	Open failure
FOR$_MIXFILACC	31	F	Mixed file access modes
FOR$_INVLOGUNI	32	F	Invalid logical unit number
FOR$_ENDFILERR	33	F	ENDFILE error
FOR$_UNIALROPE	34	F	Unit already open

[1]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. Use the fifth argument of the call to ERRSNS (condval) to obtain the unique system condition value that identifies the error. Refer to the *VAX FORTRAN Language Reference Manual* for more information about the ERRSNS subroutine.

Table 5–1 (Cont.): Summary of Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR$_SEGRECFOR	35	F	Segmented record format error
FOR$_ATTACCNON	36	F	Attempt to access non-existent record
FOR$_INCRECLEN	37	F	Inconsistent record length
FOR$_ERRDURWRI	38	F	Error during write
FOR$_ERRDURREA	39	F	Error during read
FOR$_RECIO_OPE	40	F	Recursive I/O operation
FOR$_INSVIRMEM	41	F	Insufficient virtual memory
FOR$_NO_SUCDEV	42	F	No such device
FOR$_FILNAMSPE	43	F	File name specification error
FOR$_INCRECTYP	44	F	Inconsistent record type
FOR$_KEYVALERR	45	F	Keyword value error in OPEN statement
FOR$_INCOPECLO	46	F	Inconsistent OPEN/CLOSE parameters
FOR$_WRIREAFIL	47	F	Write to READONLY file
FOR$_INVARGFOR	48	F	Invalid argument to FORTRAN Run-Time Library
FOR$_INVKEYSPE	49	F	Invalid key specification
FOR$_INCKEYCHG	50	F	Inconsistent key change or duplicate key
FOR$_INCFILORG	51	F	Inconsistent file organization
FOR$_SPERECLOC	52	F	Specified record locked
FOR$_NO_CURREC	53	F	No current record
FOR$_REWRITERR	54	F	REWRITE error
FOR$_DELERR	55	F	DELETE error
FOR$_UNLERR	56	F	UNLOCK error
FOR$_FINERR	57	F	FIND error
FOR$_LISIO_SYN [2]	59	F,C	List-directed I/O syntax error
FOR$_INFFORLOO	60	F	Infinite format loop

[2]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 though 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.

Table 5–1 (Cont.): Summary of Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR$_FORVARMIS [2]	61	F,C	Format/variable-type mismatch
FOR$_SYNERRFOR	62	F	Syntax error in format
FOR$_OUTCONERR [2,3]	63	E,C	Output conversion error
FOR$_INPCONERR [2]	64	F,C	Input conversion error
FOR$_OUTSTAOVE	66	F	Output statement overflows record
FOR$_INPSTAREQ	67	F	Input statement requires too much data
FOR$_VFEVALERR [2]	68	F,C	Variable format expression value error
SS$_INTOVF	70	F,C	Arithmetic trap, integer overflow
SS$_INTDIV	71	F,C	Arithmetic trap, integer zero divide
SS$_FLTOVF	72	F,C	Arithmetic trap, floating overflow
SS$_FLTOVF_F	72	F,C	Arithmetic fault, floating overflow
SS$_FLTDIV	73	F,C	Arithmetic trap, zero divide
SS$_FLTDIV_F	73	F,C	Arithmetic fault, zero divide
SS$_FLTUND	74	F,C	Arithmetic trap, floating underflow
SS$_FLTUND_F	74	F,C	Arithmetic fault, floating underflow
SS$_SUBRNG	77	F,C	Subscript out of range
MTH$_WRONUMARG	80	F	Wrong number of arguments
MTH$_INVARGMAT	81	F	Invalid argument to math library
MTH$_UNDEXP [4]	82	F,C	Undefined exponentiation
MTH$_LOGZERNEG [4]	83	F,C	Logarithm of zero or negative value
MTH$_SQUROONEG [4]	84	F,C	Square root of negative value

[2]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 though 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.

[3]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.

[4]Function return values for errors numbered 82, 83, 84, 87, 88, and 89 can be modified by means of user-written condition handlers. (See Chapter 9 for information about user-written condition handlers.)

Table 5–1 (Cont.): Summary of Run-Time Errors

FORTRAN Condition Symbol	Error Number	Severity	Message Text
MTH$_SIGLOSMAT [4]	87	F,C	Significance lost in math library
MTH$_FLOOVEMAT [4]	88	F,C	Floating overflow in math library
MTH$_FLOUNDMAT	89	F,C	Floating underflow in math library
FOR$_ADJARRDIM [5]	93	F,C	Adjustable array dimension error
INVMATKEY	94	F	Invalid key match specifier for key direction

[4]Function return values for errors numbered 82, 83, 84, 87, 88, and 89 can be modified by means of user-written condition handlers. (See Chapter 9 for information about user-written condition handlers.)

[5]If error number 93 (FOR$_ADJARRDIM) occurs and a user-written condition handler causes execution to continue, any reference to the array in question may cause an access violation.

See Table F-4 for more detailed descriptions of errors processed by the Run-Time Library.

5.2 Using the ERR and END Specifiers

When a severe error occurs during program execution, the Run-Time Library default action is to print an error message and terminate the program. You can use the ERR and END specifiers in I/O statements to override this default by transferring control to a specified point in the program. No error message is printed, and execution continues at the designated statement. For example, consider the following WRITE statement:

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

If an error occurs during execution of this statement, the Run-Time Library transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

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

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

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this OPEN statement, control transfers to statement 999.

5.3 Using the IOSTAT Specifier

You can use the IOSTAT specifier to continue program execution after an I/O error and to return information about I/O operations. It can supplement or replace the END and ERR transfers. Execution of an I/O statement containing the IOSTAT specifier suppresses printing of an error message and causes the specified integer variable, array element, or scalar field reference to become defined as one of the following:

- A value of –1 if an end-of-file condition occurs
- A value of 0 if neither an error condition nor an end-of-file condition occurs
- A positive integer value if an error condition occurs (this value is one of the FORTRAN-specific error numbers listed in Table 5–1)

Following execution of the I/O statement and assignment of an IOSTAT value, control transfers to the END or ERR statement label, if any. If there is no control transfer, normal execution continues.

You can include SYS$LIBRARY:FORSYSDEF.TLB($FORIOSDEF) in your program to obtain symbolic definitions for the values of IOSTAT. The symbolic names in this file have a form similar to the FORTRAN condition symbols:

Condition symbol	**IOSTAT value**
FOR$_error	FOR$IOS_error

Note that the values of the IOSTAT symbols are not the same as the values of the condition symbols described in Table 5–1.

The following example uses the IOSTAT specifier and the FORIOSDEF module to detect and process an OPEN error.

```
      CHARACTER*40 FILN
      INCLUDE '($FORIOSDEF)'
10    ACCEPT *, FILN
      OPEN (UNIT=1, FILE=FILN, STATUS='OLD', IOSTAT=IERR, ERR=100)
          .
          .
          .
      (process the input file)
          .
          .
          .
100   IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
           TYPE *, 'File:', FILN, 'Does not exist, enter new filename'
      ELSE IF (IERR .EQ. FOR$IOS_FILNAMSPE) THEN
          TYPE *, 'File:', FILN, 'Was bad, enter new filename'
      ELSE
          TYPE *, 'Unrecoverable error, code =', IERR
          STOP
      END IF
      GO TO 10
      END
```

Chapter 6

Using VAX FORTRAN in the Common Language Environment

VAX FORTRAN provides you with a variety of mechanisms for gaining access to procedures and system 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, VMS system services, and routines written in languages other than FORTRAN.

The terms subprogram, subroutine, and function are used throughout this chapter.

- A *subprogram* is a closed, ordered set of instructions that performs one or more specific tasks. Every subprogram has one or more entry points and, optionally, an argument list.
- *Subroutines* and *functions* are specific types of subprograms. A subroutine is a subprogram that does not return a value, and a function is a subprogram that returns a value by assigning that value to the function's identifier.

The VAX FORTRAN compiler operates within the VMS common language environment. This environment defines certain calling procedures and guidelines that allow you to call routines written in different languages or system routines from VAX FORTRAN. This chapter provides information on the VAX procedure-calling standard and how to access VAX system services. Detailed information about calling and using the RMS (Record Management Services) system services is provided in Chapter 7.

6.1 VAX Procedure-Calling Standard

Programs compiled by the VAX FORTRAN compiler conform to the standard defined for VAX procedure calls (see the manual *Introduction to VMS System Routines*). This standard prescribes how registers and the system-maintained call stack can be used, how function values are returned, how arguments are passed, and how procedures receive and return control.

When writing routines that can be called from VAX FORTRAN programs, you should give special consideration to the argument list descriptions in Section 6.1.3 and to the object code format description in Section 6.2.6.

6.1.1 Register and Stack Usage

The VAX Procedure Calling and Condition Handling Standard defines several registers and their uses, as listed in Table 6–1.

Table 6–1: VAX Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Current stack frame pointer
AP	Argument pointer
R1	Environment value (when necessary)
R0, R1	Function value return registers

By definition, any called routine can use registers R2 through R11 for computation, and the AP register as a temporary register.

In the VAX Procedure Calling and Condition Handling Standard, a *stack* is defined as a LIFO (last-in/first-out) temporary storage area that the system allocates for every user process.

The system keeps information on the stack about each routine call in the current image. That is, each time you call a routine, the system creates a structure, known as a *call frame*, on the stack.

The call frame for each active process contains the following:

- A pointer for the call frame of the previous routine call. This pointer corresponds to the frame pointer (FP).
- The argument pointer (AP) for the previous routine call.
- The storage address of the call to the routine; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC).
- The contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers when control returns to a calling routine.

When a routine completes execution, the system uses the frame pointer in the call frame for the current routine to locate the frame for the previous routine. The system then removes the call frame for the current routine from the stack.

6.1.2 Return Values of Procedures

A procedure is a VAX FORTRAN subprogram that performs one or more computations for other programs. Procedures can be either functions or subroutines. Both functions and subroutines can return values by storing them in variables specified in the argument list or in common blocks. However, a function, unlike a subroutine, can also return a value to the calling program by assigning the value to the function's name. The method that function procedures use to return values depends on the data type of the value, as summarized in Table 6–2.

Table 6–2: Function Return Values

Data Type	Return Method
$\left\{\begin{array}{l}\text{Logical}\\\text{Integer}\\\text{REAL*4}\end{array}\right\}$	General register R0
REAL*8	R0: High-order result R1: Low-order result
COMPLEX*8	R0: Real part R1: Imaginary part

Table 6–2 (Cont.): Function Return Values

Data Type	Return Method
$\left\{ \begin{array}{l} \text{REAL*16} \\ \text{COMPLEX*16} \end{array} \right\}$	An extra entry is added as the first entry of the argument list. This new first-argument entry points to the result.
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.

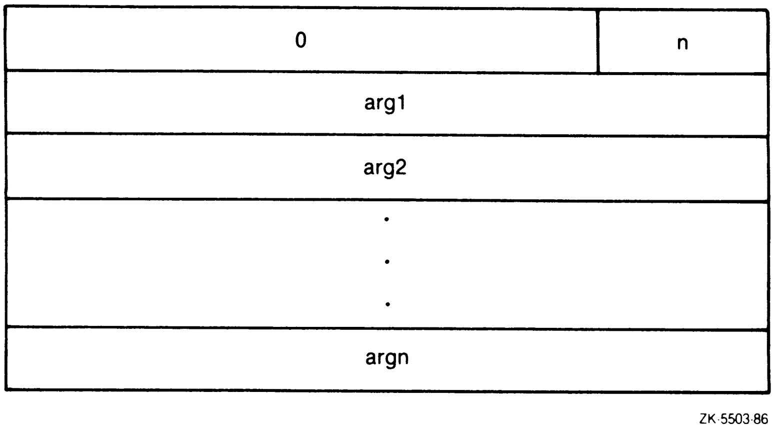
See the *VAX FORTRAN Language Reference Manual* for information on defining and invoking subprograms.

6.1.3 Argument Lists

You use an argument list to pass information to a routine and receive results.

The VAX procedure-calling standard defines an argument list as a sequence of longword (4-byte) entries. Figure 6–1 shows the structure of a typical argument list.

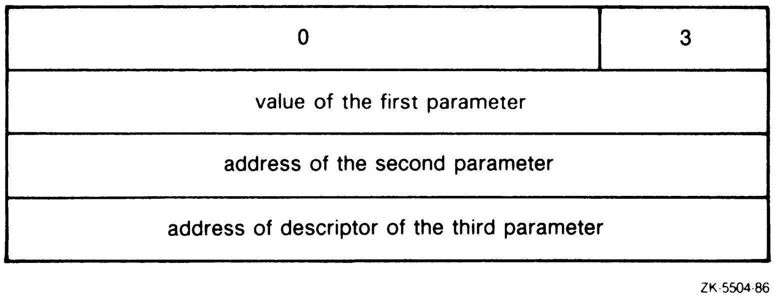
Figure 6–1: Structure of a VAX Argument List



The first longword stores the number of arguments (the argument count: *n*) as an unsigned integer value in the first byte of the longword. It must have a value between 0 and 255. It indicates how many arguments follow in the list. The upper three bytes in the argument-count entry must contain all zeros. The longwords labeled *arg1* through *argn* are the actual arguments.

The argument list contains the arguments that are passed to the subprogram. Depending on the passing mechanisms for these arguments, the forms of the arguments contained in the argument list vary. For example, if you pass three arguments, the first by value, the second by reference, and the third by descriptor, the argument list would contain the value of the first argument, the address of the second, and the address of the descriptor of the third. Figure 6–2 shows this argument list.

Figure 6–2: Example of a VAX Argument List


0
3
value of the first parameter
address of the second parameter
address of descriptor of the third parameter
ZK-5504-86

Memory for VAX FORTRAN argument lists and for VAX descriptors (generated from the use of %DESCR or by passing CHARACTER data) is usually allocated statically.

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

See Section 6.2.6.1 for examples of object code generated for VAX FORTRAN argument lists.

6.2 Argument-Passing Mechanisms

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

- By immediate value—The argument list entry contains the value.
- By reference—The argument list entry contains the address of the value.
- By descriptor—The argument list entry contains the address of a descriptor of the value.

By default, VAX FORTRAN uses the reference and descriptor mechanisms to pass arguments, depending on the argument's data type:

- The reference mechanism is used to pass all actual arguments that are numeric: logical, integer, real, and complex.
- The descriptor mechanism is used to pass all actual arguments that are character.

In some cases, a function reference or call to a non-FORTRAN procedure requires arguments in a form other than that provided by the reference and descriptor mechanisms, the VAX FORTRAN default mechanisms. Calls to VMS system services are such a case. VAX FORTRAN provides three built-in functions for passing arguments when you cannot use the default mechanisms. It also provides a built-in function for computing addresses for use in argument lists. These built-in functions are as follows:

- %VAL, %REF, %DESCR—argument list built-in functions
- %LOC—general usage built-in function

Except for the %LOC built-in function, which can be used in any arithmetic expression, these functions can appear only as unparenthesized arguments in argument lists. Note that the argument list built-in functions and %LOC built-in function are rarely used to call a procedure written in VAX FORTRAN.

The use of these functions in system service calls is described in Section 6.5.4. The sections that follow describe their use in general.

6.2.1 Passing Arguments by Reference—%REF Function

The %REF function passes the argument list entry by reference. It has the form:

```
%REF(arg)
```

The argument list entry generated by the compiler is the address of the argument (arg). The argument value can be a record name, a procedure name, or a numeric or character expression, array, or array element. In VAX FORTRAN, this is the default mechanism for passing numeric values.

6.2.2 Passing Arguments by Descripton—%DESCR Function

The %DESCR function passes the argument list entry by descriptor. It has the form:

```
%DESCR(arg)
```

The argument list entry generated by the compiler is the address of a descriptor of the argument (arg). The argument value can be any type of FORTRAN expression. The argument value cannot be a record name, record array name, or record array element. The compiler can generate VAX descriptors for all FORTRAN data types.

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

6.2.3 Passing Arguments by Immediate Value—%VAL Function

The %VAL function passes the argument list entry as a 32-bit immediate value. It has the form:

```
%VAL(arg)
```

The argument list entry generated by the compiler is the value of the argument (arg). Because argument list entries are longwords, the argument value must be an INTEGER, LOGICAL, or REAL*4 constant, variable, array element, or expression. If the value is a byte or word, it is sign extended to a longword. (The ZEXT intrinsic function can be used to produce a zero-extended value, rather than a sign-extended value.)

You may need to use the %VAL function when passing an address argument to a VAX FORTRAN subprogram. Address arguments can occur in argument lists passed to routines written in other languages. Using the %VAL function to pass an address argument pointing to a data item is equivalent to passing the item itself by reference.

6.2.4 Passing Addresses—%LOC Function

The %LOC built-in function computes the address of a storage element as an INTEGER*4 value. You can then use this value in an arithmetic expression.

The %LOC function is particularly useful for certain system services or non-FORTRAN procedures that may require argument data structures containing the addresses of storage elements. Note that the data structures should be declared volatile to protect them from possible optimizations. The effects of volatile declarations and the situations in which they should be used are discussed at length in Section 11.3.2.2. (See the discussion of the VOLATILE statement in the *VAX FORTRAN Language Reference Manual* for information on declaring volatile data structures.)

6.2.5 Examples of Argument Passing Built-in Functions

The following examples demonstrate the use of the argument list built-in functions.

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

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

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

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

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

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

See Section 6.2.6.2 for examples that include the generated object code.

6.2.6 Object Code Examples

The following sections present examples of VAX FORTRAN calls and their corresponding object code (as represented in MACRO).

6.2.6.1 Argument-Passing Examples

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

Example 1:

This example shows how the compiler generates an argument list for the arguments specified in the CALL statement.

VAX 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
```

The compiler can initialize the addresses of real variable X and array element J(3) because they 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:

In this example, the VAX FORTRAN source code defines a real array X, comprising 10 elements, and a character variable C, comprising 15 elements.

VAX 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,RO              ; Compute address of X(I) and
MOVAF     X-4[RO],ARGLST+4  ; store it in the argument list
SUBL3     #1,J,RO           ; Compute the length of C(J:K)
SUBL3     RO,K,R1           ;
MOVW      R1,L$1            ; Store length of C(J:K) in argument
                            ; list
MOVAB     C[RO],L$1+4       ; Store base address of C(J:K) in
                            ; argument list
CALLG     ARGLST,SUB        ; Call subroutine SUB
```

The actual arguments passed to subroutine SUB are the Ith element of array X and the substring C(J:K). The compiler generates an argument list consisting of three longwords: the first is the count and the next two are the address of the Ith element of X and the address of the descriptor of substring C(J:K). Note that the addresses of X(I) and C(J:K) and the length of C(J:K) are initialized to zero because these values are unknown at compile time.

6.2.6.2 Examples of Argument List Built-In Functions

The following examples show the VAX FORTRAN source code, followed by the generated object code.

Example 1: %VAL

VAX FORTRAN Source Code:

```
CALL SUB(4,%VAL(6),%VAL(-1),%VAL(ZEXT(-1)))
```

Object Code:

```
ARGLST:     .LONG 4           ; Count
            .ADDR CON4        ; Address of constant
            .LONG 6           ; value
            .LONG FFFFFFFF    ; Sign-extended value
            .LONG 0000FFFF    ; Zero-extended value

CON4:       .LONG 4
```

In this example, the compiler generates an address for the constant 4 in the first entry, but generates the actual value (6) in the following entry. Note that the constants are placed in read-only storage so that any attempt to change the value of a constant causes an access violation.

Example 2: %REF

VAX 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
```

In this example, 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

VAX 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 and the second entry contains a pointer to a descriptor.

6.2.6.3 Character Function Example

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

VAX 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).

6.3 VMS System Routines

System routines are VMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that VAX FORTRAN supports the data structures required to call the routine. The system routines used most often are VMS Run-Time Library routines and system services. System routines are documented in detail in the *VMS Run-Time Library Routines Volume* and the *VMS System Services Reference Manual.*

6.3.1 VMS Run-Time Library Routines

The VMS Run-Time Library is a library of commonly-used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular VMS Run-Time Library facility. Table 6–3 lists all of the language-independent Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 6–3: Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control DIGITAL's DECtalk device
LIB$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations
OTS$	General-purpose routines that perform tasks such as data type conversions as part of a compiler's generated code
SMG$	Screen-management routines that are used in designing, composing, and keeping track of complex images on a video screen
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings
PPL$	Parallel processing routines that help you implement concurrent programs on single-CPU and multiprocessor systems

6.3.2 VMS System Services Routines

System services are system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the VMS Run-Time Library routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). However, these services are logically divided into groups that perform similar tasks. Table 6–4 describes these groups.

Table 6–4: System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of ASTs
Change Mode	Changes the access mode of particular routines
Condition Handling	Designates condition handlers for special purposes
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters
Information	Returns information about the system, queues, jobs, processes, locks, and devices
Input/Output	Performs I/O directly, without going through VAX RMS
Lock Management	Enables processes to coordinate access to shareable system resources
Logical Names	Provides methods of accessing and maintaining pairs of character string logical names and equivalence names
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data
Process Control	Creates, deletes, and controls execution of processes
Security	Enhances the security of VMS systems
Time and Timing	Schedules events, and obtains and formats binary time valuds

6.4 Calling Routines—General Considerations

The basic steps for callinf routines are the same whether you are calling a routine written in VAX FORTRAN, a routine written in some other VAX language, a system service, or a VMS Run-Time Library routine.

To call a subroutine, you use the CALL statement. To call a function, you reference the function name in an assignment statement or as an argument in another routine call. In any case, you must specify the name of the routine being called and all arguments required for that routine. Make sure the data types and passing mechanisms for the actual arguments you are passing coincide with those declared in the routine.

If you do not want to specify a value for a required parameter, you can pass a null argument by inserting a comma (,) as a placeholder in the argument list. If you use any passing mechanism other than the default, you must specify the passing mechanism in the CALL statement or the function call.

6.5 Calling VMS System Services

You can invoke system services in a VAX FORTRAN program with a function reference or a subroutine CALL statement that specifies the system service you want to use. To specify a system service, use the form:

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

You pass arguments to the system services according to the requirements of the particular service you are calling; the service may require an immediate value, an address, the address of a descriptor, or the address of a data structure. Section 6.5.4 describes the VAX FORTRAN syntax rules for each of these cases. See the *VMS System Services Reference Manual* for a full definition of individual services.

The basic steps for calling system services are the same as those for calling any external routine. However, when calling system services (or Run-Time Library routines), additional information is often required. The sections that follow describe these requirements.

6.5.1 Obtaining Values for System Symbols

VMS uses symbolic names to identify return status values, condition values, and function codes for system services:

- Return status values are used for testing the success of system service calls.
- Condition values are used for error recovery procedures (see Chapter 9).
- Function codes are the symbolic values used as input arguments to system service calls.

The values chosen determine the specific action desired of the service.

The *VMS System Services Reference Manual* describes the symbols that are used with each system service. The *VMS I/O User's Reference Volume* describes the symbols that are used with I/O-related services.

The VAX FORTRAN symbolic definition library FORSYSDEF contains VAX FORTRAN source definitions for related groups of system symbols. Each related group of system symbols is stored in a separate text module; for example, the module $IODEF in FORSYSDEF contains PARAMETER statements that define the I/O function codes.

The modules in FORSYSDEF correspond to the symbolic definition macros that VMS MACRO programmers use to define system symbols. The modules have the same names as the macros and contain VAX FORTRAN source code, which is functionally equivalent to the MACRO source code. To determine whether you need to include other symbol definitions for the system service you want to use, refer to the documentation for that particular system service. If the documentation states that values are defined in a macro, you must include those symbol definitions in your program.

For example, the description for the *flags* argument in the SYS$MGBLSC (Map Global Section) system service states that "Symbolic names for the flag bits are defined by the $SECDEF macro." Therefore, when you call SYS$MGBLSC, you must include the definitions provided in the $SECDEF macro.

Note that the module $SYSSRVNAM in FORSYSDEF contains declarations for all system-service names. It contains the necessary INTEGER and EXTERNAL declarations for the system-service names. (The module also contains comments describing the arguments for each of the system services.) In addition, note that module $SSDEF contains system-service return status codes and is generally required whenever you access any of the services.

The modules in FORSYSDEF contain definitions for constants, bit masks, and data structures. See Section 6.5.4.4 for a description of how to create data arguments in VAX FORTRAN. Refer to Appendix B for a list of modules that are in FORSYSDEF.

You can access the modules in the FORSYSDEF library with the INCLUDE statement, using the following format:

```
INCLUDE '(module-name)'
```

The notation module-name represents the name of a module contained in FORSYSDEF. The library FORSYSDEF is searched if the specified module was not found in a previously searched library.

6.5.2 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 call to a subroutine.

For example:

```
INCLUDE '($SSDEF)'
INCLUDE '($SYSSRVNAM)'
INTEGER*2 CHANNEL
    .
    .
    .
MBX_STATUS = SYS$CREMBX(,CHANNEL,,,,,'MAILBOX')
IF (MBX_STATUS .NE. SS$_NORMAL) GO TO 100
```

In this example, the system service referenced is the Create Mailbox system service. An INTEGER*2 variable (CHANNEL) is declared to receive the channel number.

The function reference allows a return status value to be stored in the variable MBX_STATUS, which can then be checked for correct completion on return. If the function's return status is not SS$_NORMAL, failure is indicated and control is transferred to statement 100. At that point, some form of error processing can be undertaken.

You can also test the return status of a system service as a logical value. The status codes are defined so that when they are tested as logical values, successful codes have the value true and error codes have the value false. Thus, the fourth line in the example above could be changed to the following:

```
IF (.NOT. MBX_STATUS) GO TO 100
```

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

6.5.3 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 the appropriate arguments to it, as follows:

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

This call corresponds to the function reference described in Section 6.5.2. 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 whenever it is anticipated that the service could fail for any reason.

6.5.4 Passing Arguments to System Services

The description of each system service in the *VMS System Services Reference Manual* specifies the argument-passing method for each argument. Four methods are supported:

- By immediate value
- By address—this is the VAX FORTRAN default and is termed "by reference"
- By descriptor—this is the VAX FORTRAN default for CHARACTER arguments
- By data structure

These methods are discussed separately in Sections 6.5.4.1 through 6.5.4.4.

You can determine the arguments required by a system service from the service description in the *VMS System Services Reference Manual.* Each system service description indicates the service name, the number of arguments required, and the positional dependency of each argument. Table 6–5 lists the VAX FORTRAN declarations that you can use to pass any of the standard VMS data types as arguments.

Table 6–5: VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
access_bit_names	INTEGER*4(2,32) or STRUCTURE /access_bit_names/ INTEGER*4 access_name_len INTEGER*4 access_name_buf END STRUCTURE !access_bit_names RECORD /access_bit_names/ my_names(32)
access_mode	BYTE
address	INTEGER*4

address_range	INTEGER*4(2)
	or
	STRUCTURE /address_range/
	        INTEGER*4 low_address
	        INTEGER*4 high_address
	END STRUCTURE

arg_list	INTEGER*4(n)
ast_procedure	EXTERNAL
boolean	LOGICAL*4
byte_signed	BYTE
byte_unsigned	BYTE[1]
channel	INTEGER*2
char_string	CHARACTER*n
complex_number	COMPLEX*8 COMPLEX*16
cond_value	INTEGER*4

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent, provided you do not exceed the range of the signed data structure.

Table 6–5 (Cont.): VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
context	INTEGER*4
date_time	INTEGER*4(2)
device_name	CHARACTER*n
ef_cluster_name	CHARACTER*n
ef_number	INTEGER*4
exit_handler_block	STRUCTURE /exhblock/
	INTEGER*4 flink
	INTEGER*4 exit_handler_addr
	BYTE(3) /0/
	BYTE arg_count
	INTEGER*4 cond_value
	! .
	! .(optional arguments . . .
	! . one argument per longword)
	!
	END STRUCTURE !cntrlblk
	RECORD /exhblock/ myexh_block
fab	INCLUDE '($FABDEF)'
	RECORD /fabdef/ myfab
file_protection	INTEGER*4
floating_point	REAL*4
	REAL*8
	DOUBLE PRECISION
	REAL*16
function_code	INTEGER*4
identifier	INTEGER*4
io_status_block	STRUCTURE /iosb/
	INTEGER*2 iostat, !return status
	2 term_offset, !Loc. of line terminator
	2 terminator, !value of terminator
	2 term_size !size of terminator
	END STRUCTURE
	RECORD /iosb/ my_iosb

Table 6–5 (Cont.): VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
item_list_2	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2(n) (Allocate n records, where n is the number item codes plus an extra element for the end-of-list item)
item_list_3	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr,retlenadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE litmlst RECORD /itmlst/ my_itmlst_2(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)

Table 6–5 (Cont.): VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
item_list_pair	STRUCTURE /itmlist_pair/ UNION MAP INTEGER*4 code INTEGER*4 value END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst_pair RECORD /itmlst_pair/ my_itmlst_pair(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)
item_quota_list	STRUCTURE /item_quota_list/ MAP BYTE quota_name INTEGER*4 quota_value END MAP MAP BYTE end_quota_list END MAP END STRUCTURE !item_quota_list
lock_id	INTEGER*4
lock_status_block	STRUCTURE/lksb/ INTEGER*2 cond_value INTEGER*2 unused INTEGER*4 lock_id BYTE(16) END STRUCTURE !lock_status_lock
lock_value_block	BYTE(16)
logical_name	CHARACTER*n
longword_signed	INTEGER*4
longword_unsigned	INTEGER*4[1]
mask_byte	INTEGER*1

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent, provided you do not exceed the range of the signed data structure.

Table 6–5 (Cont.): VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
mask_longword	INTEGER*4
mask_quadword	INTEGER*4(2)
mask_word	INTEGER*2
null_arg	%VAL(0)
octaword_signed	INTEGER*4(4)
octaword_unsigned	INTEGER*4(4)[1]
page_protection	INTEGER*4
procedure	INTEGER*4
process_id	INTEGER*4
process_name	CHARACTER*n
quadword_signed	INTEGER*4(2)
quadword_unsigned	INTEGER*4(2)[1]

rights_holder	INTEGER*4(2)
	or
	STRUCTURE /rights_holder/
	        INTEGER*4 rights_id
	        INTEGER*4 rights_mask
	END STRUCTURE !rights_holder

rights_id	INTEGER*4
rab	INCLUDE '($RABDEF)' RECORD /rabdef/ myrab
section_id	INTEGER*4(2)
section_name	CHARACTER*n
system_access_id	INTEGER*4(2)
time_name	CHARACTER*23
uic	INTEGER*4
user_arg	Any longword quantity
varying_arg	INTEGER*4
vector_byte_signed	BYTE(n)

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent, provided you do not exceed the range of the signed data structure.

Table 6-5 (Cont.): VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
vector_byte_unsigned	BYTE(n)[1]
vector_longword_signed	INTEGER*4(n)
vector_longword_unsigned	INTEGER*4(n)[1]
vector_quadword_signed	INTEGER*4(2, n)
vector_quadword_unsigned	INTEGER*4(2,n)[1]
vector_word_signed	INTEGER*2(n)
vector_word_unsigned	INTEGER*2(n)[1]
word_signed	INTEGER*2(n)
word_unsigned	INTEGER*2(n)[1]

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent, provided you do not exceed the range of the signed data structure.

Many arguments to system services are optional. However, if you omit an optional argument, you must include a comma (,) to indicate the absence of that argument. For example, the SYS$TRNLNM system service takes five arguments. If you omit the last two arguments, you must include commas to indicate their existence, as follows:

```
ISTAT = SYS$TRNLNM(,'LNM$FILE_DEV','LOGNAM',,)
```

An invalid reference results if you specify the arguments as follows:

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

This reference provides only three arguments, not the required six.

When you omit an optional argument, the compiler supplies a default value of zero.

6.5.4.1 Immediate Value Arguments

Use value arguments when the description of the system service specifies that the argument is a "number," "mask," "mode," "value," "code," or "indicator." You must use the %VAL built-in function (see Section 6.2.3) whenever this method is required.

Immediate value arguments are used for input arguments only.

6.5.4.2 Address Arguments

Use address arguments when the description of the system service specifies that the argument is "the address of." (However, refer to Section 6.5.4.3 to determine what to do when "the address of a descriptor" is specified.) In VAX FORTRAN, this argument-passing method is called "by reference." Because this method is the VAX FORTRAN default for passing numeric arguments, you need to specify the %REF built-in function only when the data type of the argument is not logical, integer, real, or complex.

The argument description also gives the hardware data type required.

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 HANDLER3, include the following statement in the declarations portion of your program:

```
EXTERNAL HANDLER3
```

This specification defines the address of the routine for use as an input argument.

Address arguments are used for both input and output.

- 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 6–6 lists the variable data type requirements for both input and output arguments.
- 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 are not overwritten on output. The variable, therefore, does not contain the data you expect.

To store output produced by system services, you must allocate sufficient space to contain the output. You make this allocation by declaring variables of the proper size. For an illustration, refer to the Translate Logical Name system service example in Section 6.5.4.3. This service returns the length of the equivalent name string as a 2-byte value.

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

```
INCLUDE '($SYSSRVNAM)'
INTEGER*4  SYSTIM(2)
     .
     .
     .
ISTAT = SYS$GETTIM(SYSTIM)
```

The type declaration INTEGER*4 SYSTIM(2) establishes a vector consisting of two longwords, which are then used to store the time value.

Table 6–6: Variable Data Type Requirements

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	
Character string descriptor	CHARACTER*n	CHARACTER*n
Entry mask or routine	EXTERNAL	

6.5.4.3 Descriptor Arguments

Descriptor arguments are used for input and output of character strings. Use a descriptor argument when the argument description specifies "address of a character string descriptor." Because this method is the VAX FORTRAN default for character arguments, you need to specify the

%DESCR built-in function only when the data type of the argument is not character.

On input, a character constant, variable, array element, or expression is passed to the system service by descriptor. On output, two items are needed:

- The character variable or array element to hold the output string
- An INTEGER*2 variable that is set to the actual length of the output string

Thus, in the following example of the Translate Logical Name system service (SYS$TRNLNM), the logical name LOGNAM is translated to its associated name or file specification, and the output string and string length are stored in the variables EQV_BUFFER and W_NAMELEN, respectively:

```
INCLUDE '($LNMDEF)'
INCLUDE '($SYSSRVNAM)'

STRUCTURE /LIST/
  INTEGER*2 BUF_LEN/255/
  INTEGER*2 ITEM_CODE/LNM$_STRING/
  INTEGER*4 TRANS_LOG
  INTEGER*4 TRANS_LEN
  INTEGER*4 END_ENTRY/0/
END STRUCTURE    !LIST

CHARACTER*255 EQV_BUFFER
INTEGER*2 W_NAMELEN

RECORD/LIST/ ITEM_LIST
ITEM_LIST.TRANS_LOG = %LOC(EQV_BUFFER)
ITEM_LIST.TRANS_LEN = %LOC(W_NAMELEN)

ISTAT = SYS$TRNLNM(, 'LNM$FILE_DEV', 'FOR$SRC', ,ITEM_LIST)
IF (ISTAT) PRINT *, EQV_BUFFER(:W_NAMELEN)
END
```

6.5.4.4 Data Structure Arguments

Data structure arguments are used when the argument description specifies "address of a list," "address of a control block," or "address of a vector." The data structures required for these arguments are constructed in VAX FORTRAN with structure declarations blocks and the RECORD statement. The storage declared by a RECORD statement is allocated in exactly the order given in the structure declaration, with no space between adjacent items. For example, the item list required for the SYS$GETJPI system service requires a sequence of items of two words and two longwords each. By declaring each item as part of a structure, you ensure that the fields and items are allocated contiguously:

```
STRUCTURE /GETJPI_STR/
        INTEGER*2 BUFLEN, ITMCOD
        INTEGER*4 BUFADR, RETLEN
END STRUCTURE
. . .
RECORD /GETJPI_STR/ LIST(5)
```

If a given field is provided as input to the system service, the calling program must fill the field before the system service is called. You can accomplish this with data initialization (for fields with values that are known at compile time) and with assignment statements (for fields that must be computed).

When the data structure description requires a field that must be filled with an address value, use the %LOC built-in function to generate the desired address (see Section 6.2.4). When the description requires a field that must be filled with a symbolic value (system-service function code), you can define the value of the symbol by the method described in Section 6.5.1.

6.5.4.5 Examples of Passing Arguments

Example 6-1 shows a complete subroutine that uses a data structure argument to the SYS$GETJPI system service.

Example 6-1: Subroutine Using a Data Structure Argument

```
C     Subroutine to obtain absolute and incremental values of
C     process parameters:
C     CPU time, Buffered I/O count, Direct I/O count, Page faults.

      SUBROUTINE PROCESS_INFO(ABS_VALUES, INCR_VALUES)

C     Set up implicit data types so that data types indicate sizes

      IMPLICIT INTEGER*2(W), INTEGER*4(L)

C     Define the symbolic values used in the GETJPI call

      INCLUDE '($JPIDEF)'
      INCLUDE '($SYSSRVNAM)'

C     Declare the arguments and working storage

      INTEGER*4 ABS_VALUES(4), INCR_VALUES(4), LCL_VALUES(4)

C     Declare the SYS$GETJPI item list data structure in a structure declaration

      STRUCTURE /GETJPI_STR/
         INTEGER*2  BUFLEN /4/, ITMCOD /0/
         INTEGER*4  BUFADR, RETLEN /0/
      END STRUCTURE

C     Create a record with the fields defined in the structure declaration

      RECORD /GETJPI_STR/ LIST(5)

C     Assign all static values in the item list

      LIST(1).ITMCOD = JPI$_CPUTIM
      LIST(2).ITMCOD = JPI$_BUFIO
      LIST(3).ITMCOD = JPI$_DIRIO
      LIST(4).ITMCOD = JPI$_PAGEFLTS

C     Assign all item fields requiring addresses

      LIST(1).BUFADR = %LOC(LCL_VALUES(1))
      LIST(2).BUFADR = %LOC(LCL_VALUES(2))
      LIST(3).BUFADR = %LOC(LCL_VALUES(3))
      LIST(4).BUFADR = %LOC(LCL_VALUES(4))

C     Perform the system service call

      CALL SYS$GETJPI(,,,LIST,,,)
```

Example 6-1 Cont'd. on next page

Example 6–1 (Cont.): Subroutine Using a Data Structure Argument

```
C    Assign the new values to the arguments

     DO I=1,4
        INCR_VALUES(I) = LCL_VALUES(I) - ABS_VALUES(I)
        ABS_VALUES(I)  = LCL_VALUES(I)
     END DO
     RETURN
     END
```

Example 6–2 is an example of the typical use of an I/O system service. The program invokes SYS$QIOW to enable CTRL/C trapping. When the program runs, it prints an informational message whenever it is interrupted by a CTRL/C, and then it continues execution.

Example 6–2: CTRL/C Trapping Example

```
     PROGRAM TRAPC
     INCLUDE '($SYSSRVNAM)'
     INTEGER*2 TT_CHAN
     COMMON TT_CHAN
     CHARACTER*40 LINE

C    Assign the I/O channel.  If unsuccessful stop
C    otherwise initialize the trap routine.

     ISTAT = SYS$ASSIGN ('TT',TT_CHAN,,)
     IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
     CALL ENABLE_CTRLC

C    Read a line of input and echo it

10   READ (5,'(A)',END=999) LINE
     TYPE *, 'LINE READ: ', LINE
     GO TO 10
999  END

     SUBROUTINE ENABLE_CTRLC
     INTEGER*2 TT_CHAN
     COMMON TT_CHAN
     EXTERNAL CTRLC_ROUT

C    Include I/O symbols

     INCLUDE '($IODEF)'
     INCLUDE '($SYSSRVNAM)'
```

Example 6–2 Cont'd. on next page

Example 6–2 (Cont.): CTRL/C Trapping Example

```
C     Enable CTRL/C trapping and specify CTRLC_ROUT
C     as routine to be called when CTRL/C occurs

      ISTAT = SYS$QIOW( ,%VAL(TT_CHAN),
     1        %VAL(IO$_SETMODE .OR. IO$M_CTRLCAST),
     1        ,,,CTRLC_ROUT,,%VAL(3),,,)
      IF (.NOT. ISTAT) CALL LIB$STOP(%VAL(ISTAT))
      RETURN
      END

      SUBROUTINE CTRLC_ROUT
      PRINT *, 'CTRL-C pressed'
      CALL ENABLE_CTRLC
      RETURN
      END
```

Chapter 7
Using VMS Record Management Services

This chapter describes how to call VMS Record Management Services (RMS) directly from VAX FORTRAN programs. RMS is used by all utilities and VAX native-mode languages for their I/O processing. In this way, all of these utilities and user programs written in native mode languages can access files efficiently, flexibly, with device independence, and taking full advantage of the capabilities of the underlying VMS operating system.

You need to know the basic concepts concerning files on VMS systems and calling system services before reading this chapter. In particular, you should be familiar with the basic file concepts covered in the *Guide to VMS File Applications*. You also need to know the system-service calling conventions covered in Chapter 6 of this manual.

In addition, you should have access to the *VMS Record Management Services Manual*. That manual, although not written specifically for VAX FORTRAN programmers, is the definitive reference source for all information on the use of RMS.

After reading this chapter, you will be able to understand the material in the *VMS Record Management Services Manual* in terms of FORTRAN concepts and usage. You will also be able to take advantage of the material in the *Guide to VMS File Applications*, which covers more areas of RMS in greater detail than this chapter.

In particular, after reading this chapter, you should read the first two chapters in the *VMS Record Management Services Manual*; they explain many of the concepts introduced here in greater detail and provide a good introduction to RMS.

The easiest way to call RMS services directly from VAX FORTRAN is to use a USEROPEN routine. A USEROPEN routine is a subprogram that you specify in an OPEN statement. The VAX FORTRAN Run-Time Library (RTL) I/O support routines call the USEROPEN routine in place of the RMS services at the time a file is first opened for I/O.

The advantage of using a USEROPEN routine is that the VAX FORTRAN RTL sets up the RMS data structures on your behalf with initial field values that are based on parameters specified in your OPEN statement. This initialization usually eliminates most of the code needed to set up the proper input to RMS Services. As a result, you can use USEROPEN routines to take advantage of almost all of the power of RMS without most of the declarations and initialization code normally required. Section 7.2 describes how to use USEROPEN routines and gives examples. You should be familiar with the material in Section 7.1 before reading Section 7.2.

7.1 RMS Data Structures and Services

This section introduces the RMS data structures and services and describes how to use them in VAX FORTRAN programs. The first subsection describes the RMS data structures and how to declare and use them. The second subsection describes the RMS system services and how to call them from VAX FORTRAN.

7.1.1 RMS Data Structures

RMS system services have so many options and capabilities that it is impractical to use anything other than several large data structures to provide their arguments. You should become familiar with all of the RMS data structures before using RMS system services.

The RMS data structures are as follows:

- File Access Block (FAB)—used to describe files in general.
- Record Access Block (RAB)—used to describe the records in files.
- Name Block (NAM)—used to give supplementary information about the name of files beyond that provided with the FAB.
- Extended Attributes Blocks (XABs)— a family of related blocks that are linked to the FAB to communicate to RMS any file attributes beyond those expressed in the FAB.

The RMS data structures are used both to pass arguments to RMS services and to return information from RMS services to your program. In particular, an auxiliary structure, such as a NAM or XAB block, is commonly used explicitly to obtain information optionally returned from RMS services.

The *VMS Record Management Services Manual* describes how each of these data structures is used in calls to RMS services. In this section, a brief overview of each block is given, describing its purpose and how it is manipulated in VAX FORTRAN programs.

In general, there are six steps to using the RMS control blocks in calls to RMS system services:

1. Declare the structure of the blocks and the symbolic parameters used in them by including the appropriate definition modules from the FORTRAN-supplied default library FORSYSDEF.TLB.
2. Declare the memory allocation for the blocks that you need with a RECORD statement.
3. Declare the system service names by including the module $SYSSRVNAM from FORSYSDEF.TLB.
4. Initialize the values of fields needed by the service you are calling. In MACRO, facilities are provided to initialize most of the fields in the RMS data structures to reasonable default values. This default initialization capability is not provided for FORTRAN programs. The structure definitions provided for these blocks in the FORSYSDEF modules provide only the field names and offsets needed to reference the RMS data structures. You must assign all of the field values explicitly in your VAX FORTRAN program.

   Two fields of each control block are mandatory; they must be filled in with the correct values before they are used in any service call. These are the block id (BID, or COD in the case of XABs) and the block length (BLN). These are checked by all RMS services to ensure that their input blocks have proper form. These fields are initialized automatically by the appropriate declaration macro for VAX MACRO users but must be assigned explicitly in your VAX FORTRAN programs, unless you are using the control blocks provided by the FORTRAN RTL I/O routines, which initialize all control block fields. See Table 7–1 for a list of the control field values provided by the FORTRAN RTL I/O routines.
5. Invoke the system service as a function reference, giving the control blocks as arguments according to the specifications in the RMS reference manual.

6. Check the return status to ensure that the service has completed successfully.

Steps 1-4 are described for each type of control block in Sections 7.1.1.2 to 7.1.1.5. See Section 7.1.2 for descriptions of steps 5 and 6.

7.1.1.1 Using FORSYSDEF Modules to Manipulate RMS Data Structures

The FORTRAN-supplied definition library FORSYSDEF.TLB contains the required FORTRAN declarations for all of the field offsets and symbolic values of field contents described in the *VMS Record Management Services Manual*. The appropriate INCLUDE statement needed to access these declarations for each structure is described wherever appropriate in the text that follows.

In general, you need to supply one or more RECORD statements to allocate the memory for the structures that you need. For information on manipulating VAX FORTRAN records, refer to Chapter 12. See the *VMS Record Management Services Manual* for a description of the naming conventions used in RMS service calls. Only the convention for the PARAMETER declarations is described here.

The FORSYSDEF modules contain several different kinds of PARAMETER declarations. The declarations are distinguished from each other by the letter following the dollar sign ($) in their symbolic names. Each is useful in manipulating field values, but the intended use of the different kinds of PARAMETER declarations is as follows:

- Declarations that define only symbolic field values are identified by the presence of a "C_" immediately after the block prefix in their names. For example, the RAB$B_RAC field has three symbolic values, one each for sequential, keyed, and RFA access modes. The symbolic names for these values are RABC_SEQ, RABC_KEY, and RAB$C_RFA. You use these symbolic field values in simple assignment statements. For example:

```
INCLUDE '($RABDEF)'
RECORD /RABDEF/ MYRAB
 . . .
MYRAB.RAB$B_RAC = RAB$C_SEQ
 . . .
```

- Declarations that use mask values instead of explicit values to define bit offsets are identified by the presence of "M_" immediately after the block prefix in their names. For example, the FAB$L_FOP field is an INTEGER*4 field with the individual bits treated as flags. Each flag has a mask value for specifying a particular file processing option. For instance, the MXV bit specifies that RMS should maximize the version number of the file when it is created. The mask value associated with this bit has the name FAB$M_MXV. In order to use these parameters, you must use .AND. and .OR. to turn off and on specific bits in the field without changing the other bits. For example, to set the MXV flag in the FOP field, you would use the following program segment:

```
INCLUDE '($FABDEF)'
RECORD /FABDEF/ MYFAB
 . . .
MYFAB.FAB$L_FOP = MYFAB.FAB$L_FOP .OR. FAB$M_MXV
```

- Two types of declarations that define symbolic field values are also used to define flag fields within a larger named field. These are identified by the presence of "S_" or "V_" immediately after the block prefix in their names. The "S_" form of the name defines the size of that flag field (usually the value 1, for single bit flag fields), and the "V_" form defines the bit offset from the beginning of the larger field. These forms of the names can be used with the symbolic bit manipulation functions to set or clear the fields without destroying the other flags. Thus, performing the same operation as the previous example using the "V_" and "S_" flags would be done as follows:

```
INCLUDE '($FABDEF)'
RECORD /FABDEF/ MYFAB
 . . .
MYFAB.FAB$L_FOP = IBSET(MYFAB.FAB$L_FOP,FAB$V_MXV)
 . . .
```

For most of the FAB, RAB, NAM, and XAB fields that are not supplied using symbolic values, you will need to supply sizes or pointers. For the sizes, you can use ordinary numeric constants or other numeric scalar quantities. For instance, to set the maximum record number into the FAB$L_MRN field, you could use the following statement:

```
MYFAB.FAB$L_MRN = 5000
```

To supply the required pointers, usually from one block to another, you must use the %LOC built-in function to retrieve addresses. For example, to fill in the FAB$L_NAM field in a FAB block with the address of the NAM block that you want to use, you can use the following program fragment:

```
INCLUDE '($FABDEF)'
INCLUDE '($NAMDEF)'
 . . .
RECORD /FABDEF/ MYFAB, /NAMDEF/ MYNAM
 . . .
MYFAB.FAB$L_NAM = %LOC(MYNAM)
```

7.1.1.2 The File Access Block

The File Access Block (FAB) is used for calling the following services:

SYS$CLOSE	SYS$OPEN
SYS$CREATE	SYS$PARSE
SYS$DISPLAY	SYS$REMOVE
SYS$ENTER	SYS$RENAME
SYS$ERASE	SYS$SEARCH
SYS$EXTEND	

The purpose of the FAB is to describe the file being manipulated by these services. In addition to the fields that describe the file directly, there are pointers in the FAB structure to auxiliary blocks used for more detailed information about the file. These auxiliary blocks are the NAM block and one or more of the XAB blocks.

To declare the structure and parameter values for using FAB blocks, include the $FABDEF module from FORSYSDEF. For example:

```
INCLUDE '($FABDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the FAB is described at length in the *VMS Record Management Services Manual.*

If you are using a USEROPEN procedure, the actual allocation of the FAB is performed by the FORTRAN Run-Time Library I/O support routines, and you only need to declare the first argument to your USEROPEN routine to be a record with the FAB structure. For example:

Calling program:

```
 . . .
EXTERNAL MYOPEN
 . . .
OPEN (UNIT=8,  . . . , USEROPEN=MYOPEN)
 . . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
INCLUDE '($FABDEF)'
 . . .
RECORD /FABDEF/ FABARG
 . . .
```

Usually, you need to declare only one FAB block. In some situations, however, you need to use two different FAB blocks. For example, the SYS$RENAME service requires two FAB blocks, one to describe the old file name and one to describe the new file name. In any of these cases, you can declare whatever FAB blocks you need with a RECORD statement. For example:

```
INCLUDE '($FABDEF)'
 . . .
RECORD /FABDEF/ OLDFAB, NEWFAB
```

If you use any of the above service calls without using a USEROPEN routine, you must initialize the required FAB fields in your program. The FAB fields required for each RMS service are listed in the descriptions of the individual services in the *VMS Record Management Services Manual.* In addition, most services fill in output values in the FAB or one of its associated blocks. These output fields are also described with the service descriptions.

In the example programs supplied in the *VMS Record Management Services Manual,* these initial field values are described as they would be used in MACRO programs, where the declaration macros allow initialization arguments. Thus, in each case where the MACRO example shows a field being initialized in a macro, you must have a corresponding initialization at run time in your program.

For example, the *VMS Record Management Services Manual* contains an example that shows the use of the ALQ parameter for specifying the initial allocation size of the file in blocks:

```
       .TITLE CREAT  -  SET CREATION DATE
;      Program that uses XABDAT and XABDAT_STORE
;
       .PSECT LONG  WRT,NOEXE
;
MYFAB: $FAB ALQ=500, FOP=CBT, FAC=<PUT>, -
            FNM=<DISK$:[PROGRAM]SAMPLE_FILE.DAT>, -
            ORG=SEQ, RAT=CR, RFM=VAR, SHR=<NIL>, MRS=52, XAB=MYXDAT
   .
   .
   .
```

As described in the section on the XAB$L_ALQ field (in the same manual), this parameter sets the FAB field FAB$L_ALQ. This means that to perform the same initialization in VAX FORTRAN, you must supply a value to the FAB$L_ALQ field using a run-time assignment statement. For example:

```
MYFAB.FAB$L_ALQ = 500
```

The FAB$B_BID and FAB$B_BLN fields must be filled in by your program prior to their use in an RMS service call, unless they have already been supplied by the VAX FORTRAN RTL I/O routines. You should always use the symbolic names for the values of these fields, for example:

```
INCLUDE '($FABDEF)'
 . . .
RECORD /FABDEF/ MYFAB
 . . .
MYFAB.FAB$B_BID = FAB$C_BID
MYFAB.FAB$B_BLN = FAB$C_BLN
 . . .
STATUS = SYS$OPEN( . . . )
 . . .
```

7.1.1.3 The Record Access Block

The Record Access Block (RAB) is used for calling the following services:

SYS$CONNECT	SYS$READ
SYS$DELETE	SYS$RELEASE
SYS$DISCONNECT	SYS$REWIND
SYS$FIND	SYS$SPACE
SYS$FLUSH	SYS$TRUNCATE
SYS$FREE	SYS$UPDATE
SYS$GET	SYS$WAIT
SYS$NXTVOL	SYS$WRITE
SYS$PUT	

The purpose of the RAB is to describe the record being manipulated by these services. The RAB contains a pointer to the FAB used to open the file being manipulated, making it unnecessary for these services to have a FAB in their argument lists. Also, a RAB can point to only one kind of XAB, a terminal XAB.

To declare the structure and parameter values for using RAB blocks, include the $RABDEF module from FORSYSDEF. For example:

```
INCLUDE '($RABDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the RAB is described at length in the *VMS Record Management Services Manual.*

If you are using a USEROPEN procedure, the actual allocation of the RAB is performed by the VAX FORTRAN Run-Time Library I/O support routines, and you only need to declare the second argument to your USEROPEN routine to be a record with the RAB structure. For example:

Calling program:

```
 . . .
EXTERNAL MYOPEN
 . . .
OPEN (UNIT=8,  . . . , USEROPEN=MYOPEN)
 . . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
 . . .
INCLUDE '($RABDEF)'
 . . .
RECORD /RABDEF/ RABARG
 . . .
```

If you need to access the RAB used by the FORTRAN I/O system for one of the open files in your program, you can use the FOR$RAB system function. You can use FOR$RAB even if you did not use a USEROPEN routine to open the file. The FOR$RAB function takes a single argument, the unit number of the open file for which you want to obtain the RAB address. The function result is the address of the RAB for that unit.

If you use the FOR$RAB function in your program, you should declare it to be INTEGER if you assign the result value to a variable. If you do not, your program will assume that it is a REAL function and will perform an improper conversion to INTEGER.

To use the result of the FOR$RAB call, you must pass it to a subprogram as an actual argument using the %VAL built-in function. This allows the subprogram to access it as an ordinary VAX FORTRAN record argument. For example, the main program for calling a subroutine to print the RAB fields could be coded as follows:

If you need to access other control blocks in use by the RMS services for that unit, you can obtain their addresses using the link fields they contain. For example:

```
SUBROUTINE DUMPRAB(RAB)
  . . .
INTEGER*4 FABADR
INCLUDE '($RABDEF)'
RECORD /RABDEF/ RAB
  . . .
FABADR = RAB.RAB$L_FAB
  . . .
CALL DUMPFAB(%VAL(FABADR))
  . . .
```

In this example, the routine DUMPRAB obtains the address of the associated FAB by referencing the RAB$L_FAB field of the RAB. Other control blocks associated with the FAB, such as the NAM and XAB blocks, can be accessed using code similar to this example.

Usually, you need to declare only one RAB block. Sometimes, however, you may need to use more than one. For example, the multistream capability of RMS allows you to connect several RABs to a single FAB. This allows you to simultaneously access several records of a file, keeping a separate context for each record. In any case, you can declare whatever RAB blocks you need with a RECORD statement. For example:

```
INCLUDE '($RABDEF)'
  . . .
RECORD /RABDEF/ RAB1, RABARRAY(10)
```

If you use any of the above service calls without using a USEROPEN routine, you must initialize the required RAB fields in your program. The RAB fields required for each RMS service are listed in the descriptions of individual services in the *VMS Record Management Services Manual.* In addition, most services fill in output values in the RAB. These output fields are also described with the service descriptions.

In the example programs supplied in the *VMS Record Management Services Manual*, these initial field values are described as they would be used in MACRO programs, where the declaration macros allow initialization arguments. Thus, in each case where the MACRO example shows a field being initialized in a declaration macro, you must have a corresponding initialization at run time in your program.

For example, the *VMS Record Management Services Manual* contains an example that shows the use of the RAC parameter for specifying the record access mode to use:

```
        .TITLE  CREATEIDX - CREATE INDEXED FILE
        .IDENT  /V001/
    .
    .
    .
        .SBTTL  Control block and buffer storage
        .PSECT  DATA NOEXE.LONG
;
; Define the source file FAB and RAB control blocks.
;
SRC_FAB:
        $FAB    FAX=<GET>,-             ; File access for GET only
                FOP=<SQO>,-             ; DAP file transfer mode
                FNM=<SRC:>              ; Name of input file
SRC_RAB:
        $RAB    FAB=SRC_FAB,-           ; Address of associated FAB
                RAC=SEQ,-               ; Sequential record access
                UBF=BUFFER,-            ; Buffer address
                USZ=BUFFER_SIZE         ; Buffer size
    .
    .
    .
```

In the example, sequential access mode is used. As described in the section on the RAC field (in the same manual), this parameter sets the RAB$B_RAC field to the value RAB$C_SEQ. This means that to perform the same initialization in FORTRAN, you must supply RAC field values by a run-time assignment statement. For example:

```
MYRAB.RAB$B_RAC = RAB$C_SEQ
```

The RAB$B_BID and RAB$B_BLN fields must be filled in by your program prior to their use in an RMS service call, unless they have been supplied by the FORTRAN RTL I/O routines. You should always use the symbolic names for the values of these fields. For example:

```
INCLUDE '($RABDEF)'
 . . .
RECORD /RABDEF/ MYRAB
 . . .
MYRAB.RAB$B_BID = RAB$C_BID
MYRAB.RAB$B_BLN = RAB$C_BLN
 . . .
STATUS = SYS$CONNECT(MYRAB)
 . . .
```

7.1.1.4 The Name Block

The Name Block (NAM) can be used with the FAB in most FAB-related services in order to supply to or receive from RMS more detailed information about a file name. The NAM block is never given directly as an argument to an RMS service, to supply it you must link to it from the FAB. See Section 7.1.1.1 for an example of this.

To declare the structure and parameter values for using NAM blocks, include the $NAMDEF module from FORSYSDEF. For example:

```
INCLUDE '($NAMDEF)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. Each field in the NAM is described at length in the *VMS Record Management Services Manual.*

If you are using a USEROPEN procedure, the actual allocation of the NAM is performed by the VAX FORTRAN Run-Time Library I/O support routines. Because the NAM block is linked to the FAB, it is not explicitly given in the USEROPEN routine argument list. Thus, to access the NAM, you need to call a subprogram, passing the pointer by value and accessing the NAM in the subprogram as a structure. For example:

Calling program:

```
 . . .
EXTERNAL MYOPEN
 . . .
OPEN (UNIT=8,  . . .  , USEROPEN=MYOPEN)
 . . .
```

USEROPEN routine:

```
INTEGER FUNCTION MYOPEN(FABARG, RABARG, LUNARG)
  . . .
INCLUDE '($FABDEF)'
  . . .
RECORD /FABDEF/ FABARG
  . . .
CALL NAMACCESS(%VAL(FABARG.FAB$L_NAM))
  . . .
```

NAM accessing routine:

```
SUBROUTINE NAMACCESS(NAMARG)
  . . .
INCLUDE '($NAMDEF)'
  . . .
RECORD /NAMDEF/ NAMARG
  . . .
IF (NAMARG.NAM$B_ESL .GT. 132) GO TO 100
  . . .
```

Usually, you only need to declare one NAM block. You can declare whatever NAM blocks you need with a RECORD statement. For example:

```
INCLUDE '($NAMDEF)'
  . . .
RECORD /NAMDEF/ NAM1, NAM2
```

Most often, you use the NAM block to pass and receive information about the components of the file specification, such as the device, directory, file name, and file type. For this reason, most of the fields of the NAM block are CHARACTER strings and lengths. Thus, when using the NAM block, you should be familiar with the argument passing mechanisms for CHARACTER arguments described in Section 6.5.4.3.

Your program must fill in the NAM$B_BID and NAM$B_BLN fields prior to their use in an RMS service call, unless they have been supplied by the VAX FORTRAN RTL I/O routines. You should always use the symbolic names for the values of these fields. For example:

```
INCLUDE '($NAMDEF)'
  . . .
RECORD /NAMDEF/ MYNAM
  . . .
MYNAM.NAM$B_BID = NAM$C_BID
MYNAM.NAM$B_BLN = NAM$C_BLN
  . . .
```

7.1.1.5 Extended Attributes Blocks

Extended Attribute Blocks (XABs) are a family of related structures for passing and receiving additional information about files. There are nine different kinds of XABs:

- Allocation Control (XABALL)
- Date and Time (XABDAT)
- File Header Characteristics (XABFHC)
- Journaling (XABJNL)
- Key Definition (XABKEY)
- Protection (XABPRO)
- Revision Date and Time (XABRDT)
- Summary (XABSUM)
- Terminal (XABTRM)

The XABs are described in the *VMS Record Management Services Manual.* XABs are generally smaller and simpler than the FAB, RAB, and NAM blocks because each describes information about a single aspect of the file. You do not have to use all of them; for any given call to an RMS service routine, use only those that are required. Often the XAB fields override the corresponding fields in the FAB. For example, the allocation XAB describes the file's block allocation in more detail than the FAB$L_ALQ field can. For this reason, XAB$L_ALQ (the allocation field in the XABALL structure) always overrides the FAB$L_ALQ value.

The XABs used for any given RMS service call are connected to the FAB in a linked list. The head of the list is the FAB$L_XAB field in the FAB. This field contains the address of the first XAB to be used. Each successive XAB in the list links to the next using the XAB$L_NXT field. This field contains the address of the next XAB in the list. The order of the XABs in the list does not matter, but each kind of XAB must not appear more than once in the list.

The only kind of XAB that can be connected to a RAB instead of a FAB is the terminal XAB. It is linked to the RAB with the RAB$L_XAB field. This is needed because the terminal control information is dynamic and potentially changes with each record operation performed.

To declare the structure and parameter values for using the different XAB blocks, include the appropriate XAB definition module from FORSYSDEF. (The names of the XAB definition modules are listed previously in this section.) Also, because the XABs are a family of related control blocks, you also need to include the $XAB module from FORSYSDEF.TLB in order to declare the fields common to all XABs. For example, to declare the fields used in the Date and Time XAB, use the following declarations:

```
INCLUDE '($XABDAT)'
INCLUDE '($XAB)'
```

To examine the fields and values declared, use the /LIST qualifier after the right parenthesis. All of the fields in the XABs are described in detail in the *VMS Record Management Services Manual.*

If you are using a USEROPEN procedure, the actual allocation of the XABs used by the open operation is performed by the VAX FORTRAN Run-Time Library I/O support routines. Because the XAB blocks are linked to the FAB, it is not explicitly given in the USEROPEN routine argument list. Thus, to access the XABs, you need to call a subprogram and pass a pointer to it using the %VAL built-in function. For an example of this method, see Section 7.1.1.3.

To allocate space for an XAB block in your program, you need to declare it with a RECORD statement. For example:

```
INCLUDE '($XABDAT)'
INCLUDE '($XABPRO)'
 . . .
RECORD /XABDATDEF/ MYXABDAT, /XABPRODEF/ MYXABPRO
 . . .
```

For each XAB that you declare in your program, you must supply the correct COD and BLN fields explicitly. These field offsets are common to all XABs and are contained in the $XAB module in FORSYSDEF.TLB. The block id and length are unique for each kind of XAB and the symbolic values for them are contained in the separate XAB declaration modules in FORSYSDEF.TLB. For example, to properly initialize a Date and Time XAB, you could use the following code segment:

```
INCLUDE '($XABDAT)'
INCLUDE '($XAB)'
RECORD /XABDATDEF/ MYXABDAT
 . . .
MYXABDAT.XAB$B_COD = XAB$C_DAT
MYXABDAT.XAB$B_BLN = XAB$C_DATLEN
 . . .
```

7.1.2 RMS Services

In general, you need to do the same things when calling an RMS service that you need to do when calling any VMS service, that is, declare the name, pass arguments, and check status values. (See Section 6.5 for general information on calling VMS system services.) However, RMS services have some additional conventions and ease-of-use features that you should be aware of. For a more complete description of each RMS service, refer to the *VMS Record Management Services Manual*.

7.1.2.1 Declaring RMS System Service Names

As with the other system services, you should use the $SYSSRVNAM module in FORSYSDEF to declare the names of all of the RMS services. For example:

```
INCLUDE '($SYSSRVNAM)'
```

This module contains comments describing each VMS system service, including all of the RMS services, and INTEGER*4 and EXTERNAL declarations for each. Including the module allows you to use the names of the RMS services in your programs without further declaration.

7.1.2.2 Arguments to RMS Services

See Section 6.5.4 for a general discussion of passing arguments to system services.

Most RMS services require three arguments. The first is the control block to be used, generally a RAB or FAB, and is mandatory. The second and third arguments are the addresses of routines to be called if the RMS service fails or succeeds, and these are optional. Some RMS services take other arguments, but these services are rarely needed. You should always refer to the documentation for the specific service that you are calling for detailed information on its arguments.

Most RAB and FAB fields are ignored by most RMS services. The documentation of each service in the *VMS Record Management Services Manual* describes which fields are input for that service and which are output, for each control block used. Services that take a FAB as an argument are called the File Control services. Services that take a RAB as an argument are called the Record Control services. Typically, you need to use both when doing RMS I/O in your program.

In general, fields that are not documented as required for input to a service are ignored and can be left uninitialized. The exceptions are the Block Id (BID or COD) and Block Length (BLN) fields; these must always be initialized. See the preceding sections about the respective blocks for examples of how to initialize these fields.

The output of many RMS services provides the values required for input to other RMS services. For this reason, you usually only need to initialize a few fields in each block to their nondefault values. This is especially true when using RMS blocks declared with the VAX FORTRAN RTL I/O routines, for instance, when using USEROPEN routines or the FOR$RAB function.

7.1.2.3 Checking Status from RMS Services

You should always invoke RMS services as functions, rather than calling them as subroutines (see Section 6.5.2 for a general discussion of this topic). It is particularly important to check the status of RMS services because they usually do not cause an error when they fail. If the status is not checked immediately, the failure will go undetected until later in the program where it will be difficult to diagnose.

In most cases, you only need to check for success or failure by testing whether the returned status is true or false. Some services have alternate success-status possibilities. You should always check for these in cases where the program depends on the correct operation of the services. The RMS services have a unique set of status return symbols not used by any of the other VMS system services. You should always use these symbols whenever you check the individual status values returned. To obtain the declarations for these symbols, include the $RMSDEF module from FORSYSDEF.TLB. For example:

```
INCLUDE '($RMSDEF)'
```

This statement includes in your program the declarations for all of the symbolic RMS return values.

The *VMS Record Management Services Manual* documents the symbolic values, both success and failure, that can be returned from each of the services. Your program should always test each service-result status against these symbolic values and take appropriate action when a failure status is detected. You should always declare status variables as INTEGER*4 type in order to avoid unexpected numeric conversions. The recommended action depends on whether you are using RMS services in a USEROPEN routine.

In a USEROPEN routine, the VAX FORTRAN RTL I/O routines that invoke your USEROPEN routine are expecting an RMS status as an output value. For this reason, you need to return the RMS status value as the function value—for both failure and success conditions. For example:

```
INTEGER FUNCTION MYOPEN(FAB,RAB,LUN)
 . . .
INCLUDE '($SYSSRVNAM)'          ! Declare RMS service names
 . . .
MYOPEN = SYS$OPEN(FAB)
IF (.NOT. MYOPEN) RETURN
 . . .
RETURN
END
```

In this case, if the SYS$OPEN service fails, it returns an error status into the function result variable MYOPEN. If the test of MYOPEN does not indicate success, the function returns the actual RMS status as its value. Then, the RTL I/O routines will signal the appropriate FORTRAN error normally, as if a USEROPEN routine had not been used.

If the SYS$OPEN call succeeds, the program continues, and the RMS$_NORMAL success status will ultimately be returned to the FORTRAN RTL. This value will cause the OPEN statement that specifies MYOPEN to complete successfully.

However, if you are not using a USEROPEN routine, your program must indicate the error status directly, unless it is prepared to deal with it. Often, the easiest way to indicate an error and issue a helpful message is to signal the RMS condition directly with LIB$SIGNAL or LIB$STOP. For example:

```
 . . .
INCLUDE '($SYSSRVNAM)'          ! Declare RMS service names
INTEGER*4 STATUS                ! Declare a status variable
 . . .
STATUS = SYS$GET(MYRAB)
IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
```

See Chapter 9 for more information on the use of LIB$SIGNAL and LIB$STOP.

7.1.2.4 Opening a File

To perform input or output operations on a file, your program must first open the file and establish an active RMS I/O stream. To open a file, your program generally needs to call either the SYS$CREATE or SYS$OPEN services, followed by the SYS$CONNECT service. When you use an OPEN statement without a USEROPEN routine, the VAX FORTRAN RTL I/O routines do this for your program.

SYS$OPEN and SYS$CREATE provide the following file opening options:

- Use the SYS$OPEN service to open an existing file. SYS$OPEN returns an error status if the file cannot be found.
- Use the SYS$CREATE service to intentionally create a new file.
- Use SYS$CREATE with the CIF bit in the FAB$L_FOP field to open a file that may or may not exist. The SYS$CREATE service will either open the file (if it exists) or create a new one (if it does not exist). (You can use the SUP bit to force SYS$CREATE to create a new file even if one already exists.)

The value of the FAB$B_FAC field of the FAB indicates to RMS what record operations are to be done on the file being opened. If a record operation that was not indicated by the FAC field (such as a SYS$PUT) is attempted, the record service will not perform the operation and will return a failure status. This is an important file protection feature, it prevents you from accidentally corrupting a file when you use the wrong RMS service.

The SYS$CONNECT service establishes an active I/O stream, using a RAB, to a file that has been previously opened by your program. RMS identifies all active I/O streams by a unique identifier, called the Internal Stream Identifier (IFI). This value is stored in the RAB$W_ISI field of the RAB for each active stream being processed. This field must always be zero when calling SYS$CONNECT. The SYS$CONNECT service initializes this field so that subsequent operations using that RAB can be uniquely identified. Under some circumstances, you can establish more than one simultaneously active I/O stream to the same file. See the *VMS Record Management Services Manual* for more information on this topic.

7.1.2.5 Closing a File

To close a file, use the SYS$CLOSE service. This terminates all active I/O streams under way on that file and frees all RMS resources being used for processing that file. Use the SYS$DISCONNECT service if you want to end one active I/O stream, but want to continue processing the file using another stream. This service sets the RAB$W_ISI value to zero so that the RAB can be reused for another stream.

7.1.2.6 Writing Data

To write data to a file, use the SYS$PUT or SYS$WRITE service. Your program must set the PUT bit in the FAB$B_FAC field when the file is opened; otherwise, the service attempting the write operation will fail.

Use the SYS$PUT service when you want to write data in *record mode* (the default). In record mode, RMS buffers data automatically and performs the actual output operation for a whole group of records at a time. This is the mode used for all VAX FORTRAN WRITE statements. Because most programs and utilities can read data written in record mode, this mode should be used when the data being written is to be read and processed by a general program or utility.

Use the SYS$WRITE service when you want to bypass the record management capabilities of RMS and write blocks of data directly to the device without additional buffering. This mode is called *block mode* I/O and is generally much faster and uses fewer CPU resources than record mode. For this reason, it is the preferred mode for writing large amounts of unformatted data to a device. However, this mode should only be used when the program that needs to read the data can also use block mode. If the program that is to read the data cannot use block mode, you must use some other means to guarantee that the data being written can be accessed. For instance, it is not generally possible to read data written with SYS$WRITE using ordinary VAX FORTRAN READ statements. You should read the special restrictions on using block mode in the *VMS Record Management Services Manual* because SYS$WRITE may be subject to different device dependencies than SYS$PUT (record mode).

7.1.2.7 Reading Data

To read data from a file, use the SYS$GET or SYS$READ service. Your program must set the GET bit in the FAB$B_FAC field when the file is opened; otherwise, the service attempting the read operation will fail.

Use the SYS$GET service when you want to read data in *record mode* (the default). In this mode, RMS buffers data automatically and performs the actual input operation for a whole group of records at a time. This is the mode used for all VAX FORTRAN READ statements. This mode should be used whenever the program or utility that wrote the data used record mode, unless your reading program can buffer and deblock the data itself.

Use the SYS$READ service when you want to bypass the record management capabilities of RMS and read blocks of data directly from the device without buffering or deblocking. This mode is called *block mode* I/O and is generally much faster and uses fewer CPU resources than record mode. For this reason, it is the preferred mode for reading large amounts of unformatted data from a device. However, this mode should only be used when the data was written by a utility or program that wrote the data in block mode. If the file was written using record mode, RMS control information may be intermixed with the data, making it difficult to process. You should read the special restrictions on using block mode in the *VMS Record Management Services Manual* before using SYS$READ, however, because SYS$READ may be subject to different device dependencies than SYS$GET (record mode).

7.1.2.8 Other Services

RMS provides many other file and record processing services beyond just Opening, Closing, Reading and Writing. Other file processing services include the following:

- SYS$PARSE and SYS$SEARCH—process wildcard and incomplete file specifications and search for a sequence of files to be processed
- SYS$DISPLAY—retrieves file attribute information
- SYS$ENTER—inserts a file name into a directory file
- SYS$ERASE—deletes a file and removes the directory entry used to specify it
- SYS$EXTEND—increases the amount of disk space allocated to the file
- SYS$REMOVE—removes directory entries for a file

- SYS$RENAME—removes a directory entry for a file and inserts a new one in another directory

Other record processing services include the following:

- SYS$FIND—positions the record stream at the desired record for later reading or writing
- SYS$DELETE—deletes a record from the file
- SYS$SPACE—skips over one or more blocks in block I/O mode.
- SYS$TRUNCATE—truncates a file after a given record
- SYS$UPDATE—updates the value of an existing record

For complete descriptions of these and other RMS services, refer to the *VMS Record Management Services Manual.*

7.2 User-Written Open Procedures

The USEROPEN keyword in an OPEN statement provides you with a way to access RMS facilities that are otherwise not available to VAX FORTRAN programs.

The USEROPEN keyword specifies a user-written external procedure (USEROPEN procedure) that controls the opening of a file. It has the form:

```
USEROPEN = procedure-name
```

The notation procedure-name represents the symbolic name of a user-written open procedure. The procedure name must be declared in an EXTERNAL statement.

When an OPEN statement—with or without the USEROPEN keyword—is executed, the Run-Time Library uses the OPEN statement keywords to establish the RMS File Access Block (FAB) and the Record Access Block (RAB), as well as its own internal data structures. If a USEROPEN keyword is included in the OPEN statement, the Run-Time Library then calls your USEROPEN procedure instead of opening the file according to its normal defaults. The procedure can then provide additional parameters to RMS and can obtain results from RMS.

In order, the three arguments passed to a user-written open procedure by the Run-Time Library are as follows:

- The address of the FAB

- The address of the RAB
- The address of a longword containing the unit number

Using this information, your USEROPEN procedure can then perform the following operations:

- Modify the FAB and RAB (optional).
- Issue SYS$OPEN and SYS$CONNECT functions or SYS$CREATE and SYS$CONNECT functions when VAX FORTRAN I/O is to be performed (required). Your USEROPEN procedure should invoke the RMS SYS$OPEN routine if the file to be opened already exists (STATUS='OLD') or should call the RMS SYS$CREATE routine for any other file type (STATUS='NEW', 'UNKNOWN', or not specified). Note that the status value specified in the OPEN statement is not represented in either the FAB or RAB.
- Check status indicators returned by RMS services (required). Your procedure should return immediately if an RMS service returns a failure status.
- Obtain information returned by RMS in the FAB and RAB by storing FAB and RAB values in program variables (optional).
- Return a success or failure status value to the Run-Time Library (required). The RMS services SYS$CREATE, SYS$OPEN, and SYS$CONNECT return status codes. Thus, it is not necessary to set a separate status value as the procedure output if execution of one of these macros is the final step in your procedure.

For more information about the FAB and RAB, see the *VMS Record Management Services Manual.*

7.2.1 Examples of USEROPEN Routines

The following OPEN statement either creates a 1000-block contiguous file or returns an error. (The default VAX FORTRAN interpretation of the INITIALSIZE keyword is to allocate the file contiguously on a best-effort basis, but not to generate an error if the space is not completely contiguous.)

```
      EXTERNAL CREATE_CONTIG
      OPEN (UNIT=10, FILE='DATA', STATUS='NEW',
     1      INITIALSIZE=1000, USEROPEN=CREATE_CONTIG)
```

User-written open procedures are often coded in BLISS or MACRO; however, they can also be coded in VAX FORTRAN using VAX FORTRAN's record handling capability.

The following function creates a file after setting the RMS FOP bit (FAB$V_CTG) to specify contiguous allocation.

```
C       UOPEN1
C
C       Program to demonstrate the use of a simple USEROPEN routine
C

      PROGRAM UOPEN1
      EXTERNAL CREATE_CONTIG

C       OPEN the file specifying the USEROPEN routine
C
      OPEN (UNIT=10, FILE='DATA', STATUS='NEW',
     1      INITIALSIZE=1000, USEROPEN=CREATE_CONTIG)

      STOP
      END

C       CREATE_CONTIG
C
C       Sample USEROPEN function to force RMS to allocate contiguous
C       blocks for the initial creation of a file.
C
      INTEGER FUNCTION CREATE_CONTIG(FAB,RAB,LUN)

C       Required declarations
C
      INCLUDE '($FABDEF)'             ! FAB Structure
      INCLUDE '($RABDEF)'             ! RAB Structure
      INCLUDE '($SYSSRVNAM)'          ! System service name declarations
      RECORD /FABDEF/ FAB, /RABDEF/ RAB

C       Clear the "Contiguous-best-try" bit, set the "Contiguous" bit
C
      FAB.FAB$L_FOP = FAB.FAB$L_FOP .AND. .NOT. FAB$M_CBT
      FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_CTG

C       Perform the create and connect, and return status
C
      CREATE_CONTIG = SYS$CREATE(FAB)
      IF (.NOT. CREATE_CONTIG) RETURN
      CREATE_CONTIG = SYS$CONNECT(RAB)
      RETURN
      END
```

The next example shows the relationship between a VAX FORTRAN function and a USEROPEN procedure. In this case, the USEROPEN keyword on the OPEN statement specifies the name of a VAX FORTRAN procedure that permits use of the RMS variable-with-fixed-length-control (VFC) record format feature. This feature is used here to obtain a text editor's line sequence numbers that are prefixed to file records.

VAX FORTRAN program:

```
C       Function to retrieve text editor files, with line numbers
C       Example of USEROPEN keyword of OPEN statement

      INTEGER FUNCTION SOSOPEN(UNIT_NUMBER, FILENAME)

      CHARACTER FILENAME*(*)
      INTEGER*2 LINE_NUMBER(0:99)
      INTEGER*4 UNIT_NUMBER, SUCCESS_CODE
      EXTERNAL GET_CNTRL_FLD
      COMMON /LINE_SEQS/LINE_NUMBER

C       Perform the open

      OPEN (UNIT=UNIT_NUMBER, FILE=FILENAME, STATUS='OLD',
     1      USEROPEN=GET_CNTRL_FLD, IOSTAT=SUCCESS_CODE)

      SOSOPEN = SUCCESS_CODE
      RETURN
      END
```

USEROPEN procedure:

```
      INTEGER FUNCTION GET_CNTRL_FLD (FAB,RAB,LUN)
      INCLUDE '($SYSSRVNAM)'
      INCLUDE '($FABDEF)'
      INCLUDE '($RABDEF)'
      RECORD /FABDEF/ FAB
      RECORD /RABDEF/ RAB
      INTEGER*4 LUN, STATUS
      INTEGER*2 LINE_NUMBER (0:99)
      COMMON /LINE_SEQS/ LINE_NUMBER

C       Set size of header field into FAB

      FAB.FAB$B_FSZ = 2

C       Set address into RAB

      RAB.RAB$L_RHB = %LOC (LINE_NUMBER(LUN))

C       Perform the open

      STATUS = SYS$OPEN (FAB)

C       If opened ok, connect stream to file

      IF (STATUS) STATUS = SYS$CONNECT (RAB)

C       Return status
```

```
GET_CNTRL_FLD = STATUS
RETURN
END
```

The file is created with the VFC record type. It uses the 2-byte fixed control field to store a 16-bit unsigned integer. To access this control field from VAX FORTRAN, a USEROPEN procedure must tell RMS the size of the field it wants (FAB$B_FSZ) and the location of a variable in which to place the line number when a record is read (RAB$L_RHB).

The USEROPEN procedure in this example, GET_CNTRL_FLD, determines which logical unit is being opened and stores in the RAB the address of an element in the common array LINE_NUMBER.

The USEROPEN procedure then opens the file and connects the record stream. If the operation is successful, a success status is returned from GET_CNTRL_FLD. Otherwise, a failure status is returned, causing the Run-Time Library to report that the operation failed.

Each time a READ is done from the file, RMS places the line number in the appropriate array element. (The LINE_NUMBER array has 100 elements, corresponding to the logical unit numbers 0 through 99.)

7.2.2 RMS Control Structures

Use of the USEROPEN keyword has some restrictions. The Run-Time Library constructs the following RMS control structures before calling the USEROPEN procedure:

FAB	File Access Block
RAB	Record Access Block
NAM	Name Block
XAB	Extended Attributes Blocks
ESA	Expanded String Area
RSA	Resultant String Area

A USEROPEN procedure should not alter the allocation of these structures, although it can modify the contents of many of the fields. Your procedure can also add additional XAB control blocks by linking them anywhere into the XAB chain. However, you must exercise caution when changing fields that have been set as a result of VAX FORTRAN keywords, because the Run-Time Library may not be aware of the changes. For example, do not attempt to change the record size in your USEROPEN procedure; instead, use the VAX FORTRAN keyword RECL. Always use an OPEN statement keyword if one is available.

Although the FAB, RAB, and NAM blocks remain defined during the time that the unit is opened, the XAB blocks are present only until the file has been successfully opened. In addition, the locations of the ESA and RSA strings are changed after the file is opened. Therefore, your USEROPEN procedure should not store the addresses of the RMS control structures. Instead, have your program call FOR$RAB to obtain the address of the RAB once the file is opened and then access the other structures through the RAB.

NOTE

> Future releases of the Run-Time Library may alter the use of some RMS fields. Therefore, you may have to alter your USEROPEN procedures accordingly.

Table 7–1 shows which FAB, RAB, and XAB fields are either initialized before your USEROPEN procedure is called or examined upon return from your USEROPEN procedure. All fields are initialized in response to OPEN statement keywords or default to zero. Fields labeled with a hyphen (-) are initialized to zero. Fields labeled with an asterisk (*) are returned by RMS.

Table 7–1: RMS Fields Available with USEROPEN

Field Name	Description	VAX FORTRAN OPEN Keyword and Value
FAB$L_ALQ	Allocation quantity	n if INITIALSIZE=n
FAB$B_BKS	Bucket size	(BLOCKSIZE + 511)/512
FAB$W_BLS	Block size	n if BLOCKSIZE=n
FAB$L_CTX	Context	- (Reserved for future use by DIGITAL)
FAB$W_DEQ	Default file extension quantity	n if EXTENDSIZE=n
FAB$L_DEV	Device characteristics	*
FAB$L_DNA	Default file specification string address	UNIT=nn Set to FOR0nn.DAT or FORREAD.DAT, FORACCEPT.DAT, FORTYPE.DAT, or FORPRINT.DAT or to default file specification string
FAB$B_DNS	Default file specification string size	Set to length of default file specification string
FAB$B_FAC	File access	READONLY Set to 0 if READONLY (RMS default), else set to FAB$M_GET + FAB$M_PUT + FAB$M_UPD + FAB$M_TRN + FAB$M_DEL
FAB$L_FNA	File specification string address	FILE=filename if FILE present, else set to FOR0nn, FOR$READ, FOR$ACCEPT, FOR$TYPE, FOR$PRINT, SYS$INPUT, or SYS$OUTPUT
FAB$B_FNS	File specification string size	Set to length of file specification string
FAB$L_FOP	File processing options	
FAB$V_CBT	Contiguous best try	1 if INITIALSIZE=n
FAB$V_CIF	Create if nonexistent	1 if READONLY not specified and STATUS='UNKNOWN' or STATUS omitted
FAB$V_CTG	Contiguous allocation	-
FAB$V_DFW	Deferred write	1
FAB$V_DLT	Delete on close service	Set at FORTRAN close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_ESC	Escape, nonstandard processing	-
FAB$V_INP	Input, make this SYS$INPUT	-
FAB$V_KFO	Known file open	-
FAB$V_MXV	Maximize version number	-
FAB$V_NAM	Name block inputs	-
FAB$V_NEF	Not positioned at end of file	1 unless ACCESS= 'APPEND'
FAB$V_NFS	Not file structured	-

Table 7–1 (Cont.): RMS Fields Available with USEROPEN

Field Name	Description	VAX FORTRAN OPEN Keyword and Value
FAB$V_OFP	Output file parse	-
FAB$V_POS	Current position (after closed file)	-
FAB$V_PPF	Process permanent file	-
FAB$V_RCK	Read check	-
FAB$V_RWC	Rewind on close service	-
FAB$V_RWO	Rewind on open service	-
FAB$V_SCF	Submit command (when closed)	Set at FORTRAN close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_SPL	Spool to printer	Set at FORTRAN close, depending upon DISP keyword in OPEN or CLOSE, or STATUS keyword in CLOSE
FAB$V_SQO	Sequential only	1 if a network file and ACCESS='SEQUENTIAL' or 'APPEND', else 0
FAB$V_SUP	Supersede	-
FAB$V_TEF	Truncate at end-of-file	-
FAB$V_TMD	Temporary, marked for delete	1 if STATUS= 'SCRATCH', else 0
FAB$V_TMP	Temporary (file with no directory entry)	-
FAB$V_UFM	User file mode	-
FAB$V_UFO	User file open or create file only	-
FAB$V_WCK	Write check	-
FAB$B_FSZ	Fixed control area size	-
FAB$W_IFI	Internal file identifier	*
FAB$L_MRN	Maximum record number	n if MAXREC=n
FAB$W_MRS	Maximum record size	n if RECORDTYPE='FIXED' or ORGANIZATION='RELATIVE' or ='INDEXED', else 0
FAB$L_NAM	Name block address	Set to address of name block; both the expanded and resultant string areas are set up, but the related filename string is not
FAB$B_ORG	File organization	FAB$C_IDX if ORGANIZATION='INDEXED' FAB$C_REL if ORGANIZATION='RELATIVE' FAB$C_SEQ if ORGANIZATION='SEQUENTIAL' or omitted
FAB$B_RAT	Record attributes	
FAB$V_FTN	FORTRAN carriage control	1 if CARRIAGECONTROL='FORTRAN' or not specified
FAB$V_CR	Print LF and CR	1 if CARRIAGECONTROL='LIST'

Table 7–1 (Cont.): RMS Fields Available with USEROPEN

Field Name	Description	VAX FORTRAN OPEN Keyword and Value
FAB$V_BLK	Do not cross block boundaries	1 if NOSPANBLOCKS
FAB$B_RFM	Record format	FAB$C_FIX if RECORDTYPE='FIXED' FAB$C_VAR if RECORDTYPE='VARIABLE' FAB$C_VAR if RECORDTYPE='SEGMENTED' FAB$C_STM if RECORDTYPE='STREAM' FAB$C_STMCR if RECORDTYPE='STREAM_CR' FAB$C_STMLF if RECORDTYPE='STREAM_LF'
FAB$B_RTV	Retrieval window size	-
FAB$L_SDC	Spooling device characteristics	•
FAB$B_SHR	File sharing	
FAB$V_PUT	Allow other PUTs	1 if SHARED
FAB$V_GET	Allow other GETs	1 if SHARED
FAB$V_DEL	Allow other DELETEs	1 if SHARED
FAB$V_UPD	Allow other UPDATEs	1 if SHARED
FAB$V_NIL	Allow no other operations	-
FAB$V_UPI	User-provided interlock	-
FAB$V_MSE	Multistream allowed	-
FAB$L_XAB	Extended attribute block address	The XAB chain always has a File Header Characteristics (FHC) extended attribute block in order to get longest record length (XAB$W_LRL). If the KEY=keyword is specified, key index definition blocks will also be present. DIGITAL may add additional XABs in the future. Your USEROPEN procedure may insert XABs anywhere in the chain.[1]
RAB$L_BKT	Bucket code	-
RAB$L_CTX	Context	- (Reserved for future use by DIGITAL)
RAB$L_FAB	FAB address	Set to address of FAB
RAB$W_ISI	Internal stream ID	•
RAB$L_KBF	Key buffer address	Set to address of longword containing logical record number if ACCESS='DIRECT'
RAB$B_KRF	Key of reference	0
RAB$B_KSZ	Key size	-
RAB$B_MBC	Multiblock count	If BLOCKSIZE=n, use (n + 511)/512
RAB$B_MBF	Multibuffer count	n if BUFFERCOUNT=n

Table 7–1 (Cont.): RMS Fields Available with USEROPEN

Field Name	Description	VAX FORTRAN OPEN Keyword and Value
RAB$L_PBF	Prompt buffer address	-
RAB$B_PSZ	Prompt buffer size	
RAB$B_RAC	Record access mode	
RAB$C_KEY	If ACCESS='DIRECT' or 'KEYED'	
RAB$C_SEQ	If ACCESS='SEQUENTIAL' or 'APPEND', or ACCESS omitted	
RAB$C_RFA	-	
RAB$L_RBF	Record address	Set later
RAB$L_RHB	Record header buffer	-
RAB$L_ROP	Record processing options	
RAB$V_ASY	Asynchronous	-
RAB$V_BIO	Block I/O	-
RAB$V_CCO	Cancel CTRL/O	-
RAB$V_CVT	Convert to uppercase	-
RAB$V_EOF	End-of-file	1 if ACCESS='APPEND'
RAB$V_KGE	Key greater than or equal to	-
RAB$V_KGT	Key greater than	-
RAB$V_LIM	Limit	-
RAB$V_LOC	Locate mode	1
RAB$V_NLK	No lock	-
RAB$V_NXR	Nonexistent record	-
RAB$V_PMT	Prompt	-
RAB$V_PTA	Purge type-ahead	-
RAB$V_RAH	Read-ahead	1
RAB$V_RLK	Read locked record allowed	-
RAB$V_RNE	Read no echo	-
RAB$V_RNF	Read no filter	-
RAB$V_TMO	Timeout	-
RAB$V_TPT	Truncate on PUT	1
RAB$V_UIF	Update if	1 if ACCESS='DIRECT'

Table 7–1 (Cont.): RMS Fields Available with USEROPEN

Field Name	Description	VAX FORTRAN OPEN Keyword and Value
RAB$V_ULK	Manual unlocking	·
RAB$V_WBH	Write-behind	1
RAB$W_RSZ	Record size	Set later
RAB$B_TMO	Timeout period	·
RAB$L_UBF	User record area address	Set later
RAB$W_USZ	User record area size	Set later

Note that RMS does not allow multiple instances of the same type XAB. To be compatible with future releases of the Run-Time Library, your procedure should scan the XAB chain for XABs of the type to be inserted. If one is found, it should be used instead.

7.3 Example of Block Mode I/O Usage

The following example shows a complete application of calling the RMS block I/O services SYS$WRITE and SYS$READ directly from VAX FORTRAN. The example is in the form of a complete program called BIO.FOR that writes out an array of REAL*8 values to a file using SYS$WRITE, closes the file, and then reads the data back in using SYS$READ operations with a different I/O transfer size. This program consists of five routines:

BIO	Main control program
BIOCREATE	USEROPEN routine to create the file
BIOREAD	USEROPEN routine to open the file for READ access
OUTPUT	Function that actually outputs the array
INPUT	Function that actually reads the array and checks it

Main Program—BIO

```
C       BIO.FOR
C
C       Program to demonstrate the use of RMS Block I/O operations
C       from VAX FORTRAN.
C
        OPTIONS /EXTEND_SOURCE                                      ❶
        PROGRAM BIO

C       Declare the Useropen routines as external
C
        EXTERNAL BIOCREATE, BIOREAD

C       Declare status variable, functions, and unit number
C
        LOGICAL*4 STATUS, OUTPUT, INPUT
        INTEGER*4 IUN/1/

C       Open the file                                                ❷
C
        OPEN(UNIT=IUN, FILE='BIODEMO.DAT', FORM='UNFORMATTED',
       1     STATUS='NEW', RECL=128, BLOCKSIZE=512,
       1     ORGANIZATION='SEQUENTIAL', IOSTAT=IOS,
       1     ACCESS='SEQUENTIAL', RECORDTYPE='FIXED',
       1     USEROPEN=BIOCREATE, INITIALSIZE=100)

        IF (IOS .NE. 0) STOP 'Create failed'                        ❸

C       Now perform the output
C
        STATUS = OUTPUT(%VAL(FOR$RAB(IUN)))                         ❹
        IF (.NOT. STATUS) STOP 'Output failed'                      ❸

C       Close the file for output
C
        CLOSE (UNIT=IUN)

C       Confirm output complete
C
        TYPE *, 'Output complete, file closed'

C       Now open the file for input                                  ❷
C
        OPEN(UNIT=IUN, FILE='BIODEMO.DAT', FORM='UNFORMATTED',
       1     STATUS='OLD', IOSTAT=IOS, USEROPEN=BIOREAD, DISP='DELETE')

        IF (IOS .NE. 0) STOP 'Open for read failed'                 ❸

C       Now read the file back
C
        STATUS = INPUT(%VAL(FOR$RAB(IUN)))                          ❹
        IF (.NOT. STATUS) STOP 'Input failed'                       ❸

C       Success, output that all is well
C
        STOP 'Correct completion of Block I/O demo'
        END
```

Notes:

❶ The /EXTEND_SOURCE option is used to suppress the sequence number field.

❷ Most of the necessary OPEN options for the file are specified with OPEN statement parameters. This is recommended whenever an OPEN statement qualifier exists to perform the desired function because it allows the VAX FORTRAN RTL I/O processing routines to issue appropriate error messages when an RMS routine returns an error status.

Note the discrepancy between RECL and BLOCKSIZE in the first OPEN statement. Both keywords specify 512 bytes, but the number given for RECL is 128. This is because the unit implied in the RECL keyword is longwords for unformatted files.

When using Block I/O mode, the blocksize used in the I/O operations is determined by the routine that actually does the operation. Thus, the OUTPUT routine actually transfers two 512-byte blocks at a time; whereas, the INPUT routine actually transfers four 512-byte blocks at once. In general, the larger the transfers, the more efficiently the I/O is performed. The maximum I/O transfer size allowed by RMS is 65535 bytes.

❸ The error processing in this routine is very crude; the program simply stops with an indicator of where the problem occurred. In real programs, you should provide more extensive error processing and reporting functions.

❹ The function FOR$RAB is used to supply the appropriate RAB address to the OUTPUT and INPUT routines. The %VAL function is used to transform the address returned by the FOR$RAB function to the proper argument passing mechanism so that the dummy argument RAB in INPUT and OUTPUT can be addressed properly.

USEROPEN Functions—BIOCREATE and BIOREAD

Aside from the normal declarations needed to define the symbols properly, the only interesting aspect to these routines is the setting of the BIO bit in the File Access field of the FAB. This is the only condition required for block I/O. If you wish to perform both block and record I/O on the file without closing it, you need to set the BRO bit as well. For more information on mixing block and record mode I/O, see the *VMS Record Management Services Manual*. Note that the only difference between BIOCREATE and BIOREAD is the use of SYS$CREATE and SYS$OPEN services, respectively.

```
C       BIOCREATE
C
C       USEROPEN routine to set the Block I/O bit
C       and create the BLOCK I/O demo file.
C
      INTEGER FUNCTION BIOCREATE(FAB, RAB, LUN)
      INTEGER LUN

C       Declare the necessary interface names
C
      INCLUDE '($FABDEF)'
      INCLUDE '($RABDEF)'
      INCLUDE '($SYSSRVNAM)'

C       Declare the FAB and RAB blocks
C
      RECORD /FABDEF/ FAB, /RABDEF/ RAB

C       Set the Block I/O bit in the FAC (GET and PUT
C       bits set by RTL)
C
      FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_BIO

C       Now do the Create and Connect
C
      BIOCREATE = SYS$CREATE(FAB)
      IF (.NOT. BIOCREATE) RETURN
      BIOCREATE = SYS$CONNECT(RAB)
      IF (.NOT. BIOCREATE) RETURN

C       Nothing more to do at this point, just return
C
      RETURN
      END


C       BIOREAD
C
C       USEROPEN routine to set the Block I/O bit and
C       open the Block I/O demo file for reading
C
      INTEGER FUNCTION BIOREAD(FAB, RAB, LUN)
      INTEGER LUN

C       Declare the necessary interface names
C
      INCLUDE '($FABDEF)'
      INCLUDE '($RABDEF)'
      INCLUDE '($SYSSRVNAM)'

C       Declare the FAB and RAB blocks
C
      RECORD /FABDEF/ FAB, /RABDEF/ RAB
```

```
C         Set the Block I/O bit in the FAC (GET and PUT
C         bits set by RTL)
C
        FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_BIO
C
C         Now do the Open and Connect
C
        BIOREAD = SYS$OPEN(FAB)
        IF (.NOT. BIOREAD) RETURN
        BIOREAD = SYS$CONNECT(RAB)
        IF (.NOT. BIOREAD) RETURN

C         Nothing more to do at this point, just return
C
        RETURN
        END
```

OUTPUT Routine

The following routine initializes the array A and performs the SYS$WRITE operations. Beyond the normal RTL initialization, only the RSZ and RBF fields in the RAB need to be initialized in order to perform the SYS$WRITE operations. The %LOC function is used to create the address value required in the RBF field. One of the main reasons that block mode I/O is so efficient is that it avoids copy operations by using the data areas of the program directly for the output buffer. If the program specified, for a write to a disk device, a value for RSZ that was not an integral multiple of 512, the final block would be only partly filled.

```
C
C         OUTPUT
C
C         Function to output records in block I/O mode
C
        OPTIONS /EXTEND_SOURCE
        LOGICAL FUNCTION OUTPUT(RAB)

C         Declare RMS names
C
        INCLUDE '($RABDEF)'
        INCLUDE '($SYSSRVNAM)'

C         Declare the RAB
C
        RECORD /RABDEF/ RAB

C         Declare the Array to output
C
        REAL*8 A(6400)

C         Declare the status variable
C
        INTEGER*4 STATUS
```

```
C      Initialize the array
C
      DO I=6400,1,-1
         A(I) = I
      ENDDO

C      Now, output the array, two 512-byte (64 elements)
C      blocks at a time
C
      OUTPUT = .FALSE.
      RAB.RAB$W_RSZ = 1024
      DO I=0,99,2

C          For each block, set the buffer address to the
C          proper array element
C
         RAB.RAB$L_RBF = %LOC(A(I*64+1))
         STATUS = SYS$WRITE(RAB)
         IF (.NOT. STATUS) RETURN
      ENDDO

C      Successful output completion
C
      OUTPUT = .TRUE.
      RETURN
      END
```

INPUT Routine

The following routine reads the array A from the file and verifies its values. The USZ and UBF fields of the RAB are the only fields that need to be initialized. The I/O transfer size is twice as large as the OUTPUT routine. The reason that this can be done is that the OUTPUT routine writes, to a disk device, an integral number of 512-byte blocks. This method cannot be used if the writing routine either specifies an RSZ that is not a multiple of 512 or attempts to write to a magnetic tape device.

```
C
C      INPUT
C
C      Function to input records in block I/O mode
C
      OPTIONS /EXTEND_SOURCE
      LOGICAL FUNCTION INPUT(RAB)

C      Declare RMS names
C
      INCLUDE '($RABDEF)'
      INCLUDE '($SYSSRVNAM)'

C      Declare the RAB
C
      RECORD /RABDEF/ RAB
```

```
C      Declare the Array to output
C
     REAL*8 A(6400)

C      Declare the status variable
C
     INTEGER*4 STATUS

C      Now, read the array, four 512-byte (64 elements)
C      blocks at a time
C
     INPUT = .FALSE.
     RAB.RAB$W_USZ = 2048
     DO I=0,99,4

C          For each block, set the buffer address to
C          the proper array element
C
         RAB.RAB$L_UBF = %LOC(A(I*64+1))
         STATUS = SYS$READ(RAB)
         IF (.NOT. STATUS) RETURN
     ENDDO

C      Successful input completion if data is correct
C
     DO I=6400,1,-1
         IF (A(I) .NE. I) RETURN
     ENDDO

     INPUT = .TRUE.
     RETURN
     END
```

Chapter 8

Interprocess Communications

This chapter contains information on how to exchange and share data among local and remote processes. (Local processes involve a single VAX processor, and remote processes involve separate VAX processors that are interconnected by means of DECnet.)

8.1 Local Processes—Sharing and Exchanging Data

Interprocess communication mechanisms provided for local processes provide the following capabilities:

- Program image sharing in shareable image libraries
- Data sharing in installed common areas
- Data sharing in files
- Information passing by means of mailboxes
- Information passing over DECnet–VAX network links

These capabilities are discussed in the sections that follow.

8.1.1 Sharing Images in Shareable Image Libraries

If you have a routine that is invoked by more than one program, you should consider establishing it as a shareable image and installing it on your system.

Establishing a routine as a shareable image provides the following benefits:

- Saves disk space—The executable images to which the shareable image is linked do not actually include the shareable image. Only one copy of the shareable image exists.
- Simplifies maintenance—If you use transfer vectors and the GSMATCH option, you can modify, recompile, and relink a shareable image without having to relink the executable images that reference it.

Installing a shareable image as shared (INSTALL command, /SHARED qualifier) can also save memory.

The steps to creating and installing a shareable image are as follows:

1. Compile the source file containing that routine that you want to establish as a shareable image.
2. Link the shareable image object file that results from step 1, specifying any object files that contain routines referenced by the shareable image object file.

   The VMS Linker provides a variety of options that you should consider before performing the link operation. See the *VMS Linker Utility Manual* for detailed information on shareable images and linker options.
3. Create a shareable image library using the Library Utility's LIBRARY command. See the *Guide to Creating VMS Modular Procedures* for detailed information on creating shareable image libraries.
4. Install the shareable image (the results of step 3) on your system as a shared image by using the Install Utility's INSTALL command (with the /SHARED qualifier). For detailed information on how to perform this operation, see the *VMS Install Utility Manual.*

Any programs that access a shareable image must be linked with that image. When performing the link operation, you must specify one of the following items on your LINK command:

- The name of the shareable image library containing the symbol table of the shareable image. Use the /LIBRARY qualifier to identify a library file.

- A linker options file that contains the name of the shareable image file. Use the /SHAREABLE qualifier to identify a shareable image file. (If you specify the /SHAREABLE qualifier on the LINK command line and you do not specify an options file, the linker creates a shareable image of the object file you are linking.)

The resulting executable image contains the contents of each object module and a pointer to each shareable image.

8.1.2 Sharing Data in Installed Common Areas

Sharing the same data among two or more processes can be done using installed common areas.

Typically, you use an installed common area for interprocess communication or for two or more processes to access the same data simultaneously.

To communicate between processes using a common area, first install the common area as a shareable image:

1. Create the common area—Write a VAX FORTRAN program that declares the variables in the common area and defines the common area. This program should not contain executable code. For example:

   ```
   COMMON /WORK_AREA/ WORK_ARRAY(8192)
   END
   ```

2. Make it a shareable image—Compile the program containing the common area and use the LINK/SHAREABLE command to create a shareable image containing the common area.

   ```
   $ FORTRAN INC_COMMON
   $ LINK/SHAREABLE INC_COMMON
   ```

3. Install the shareable image—Invoke the interactive Install Utility. When the INSTALL> prompt appears, type the following: the CREATE command, the complete file specification of the shareable image that contains the common area (file type defaults to EXE), and the qualifiers /WRITEABLE and /SHARED. (This operation requires CMKRNL privilege.) The Install utility installs your shareable image and reissues the INSTALL> prompt. Type EXIT to exit.

   ```
   $ INSTALL
   INSTALL> CREATE DISK$USER:[INCOME.DEV]INC_COMMON/WRITEABLE/SHARED
   INSTALL> EXIT
   ```

A disk containing an installed image cannot be dismounted until you invoke the Install Utility and type DELETE, followed by the complete file specification of the image. To exit from the Install Utility, use the EXIT subcommand.

See the *VMS Install Utility Manual* for additional information about the Install Utility.

When the common area has been installed, use the following steps to access the data from any program:

1. Include the same variable definitions and common area definitions in the accessing program.
2. Compile the program.
3. Link the accessing program against the installed common area program. Use an options file to specify the common area program as a shareable image.

   LINK commands:

   ```
   $ LINK INCOME, INCOME/OPTION
   $ LINK REPORT, INCOME/OPTION
   ```

   Specification in linker options file:

   ```
   INC_COMMON/SHAREABLE
   ```

4. Execute the accessing program.

In the previous series of examples, the two programs INCOME and REPORT access the same area of memory through the installed common area WORK_AREA.

8.1.2.1 Synchronizing Access

Typically, programs accessing shared data use common event flag clusters to synchronize read and write access to the data. In the simplest case, one event flag in a common event flag cluster might indicate that a program is writing data and a second event flag in the cluster might indicate that a program is reading data. Before accessing the shared data, a program must examine the common event flag cluster to ensure that accessing the data does not conflict with an operation already in progress.

See the *VMS System Services Reference Manual* for detailed information about the use of event flags.

8.1.3 Sharing Data in Files

With the RMS file-sharing capability, you can allow file access by more than one program at a time or by the same program on more than one logical unit.

There are two kinds of file sharing: read sharing and write sharing.

- Read sharing occurs when multiple programs are reading a file at the same time.
- Write sharing takes place when at least one program is writing a file and at least one other program is either reading or writing the same file.

All three file organizations—relative, indexed, and sequential - permit read and write access to shared files.

The extent to which file sharing can take place is determined by two factors: the type of device on which the file resides and the explicit information supplied by the user. These factors have the following effects:

- *Device type*—Sharing is possible only on disk files.
- *Explicit file-sharing information supplied by accessing programs*—Whether file sharing actually takes place depends on information provided to VMS RMS by each program accessing the file. In VAX FORTRAN programs, this information is supplied by the READONLY and SHARED keywords in the OPEN statement.

  Read sharing is accomplished when READONLY is specified by all programs accessing the file. Write sharing is accomplished when the program specifies SHARED.

  Programs that specify READONLY or SHARED can access a file simultaneously, with the exception that a file opened for READONLY cannot be accessed by a program that specifies SHARED.

  If READONLY or SHARED is not specified by both the program that initially opened a file and any other program that attempts to access that file, the latter program's attempt to access the file will fail. That is, a program without a READONLY or SHARED keyword will fail in its attempt to open a file currently being accessed by some other program, just as a program specifying READONLY or SHARED will fail to open a file if the program currently accessing that file did not specify READONLY or SHARED.

When two or more programs are write sharing a file, each program should use one of the error-processing mechanisms described in Chapter 9. Use of one of these controls, the RMS record-locking facility, prevents program failure due to a record-locking error.

The RMS record-locking facility, along with the logic of the program, prevents two processes from accessing the same record at the same time. Record locking ensures that a program can add, delete, or update a record without having to check whether the same record is simultaneously being accessed by another process.

When a program opens a relative, sequential, or indexed file specifying SHARED, RMS locks each record as it is accessed. When a record is locked, any program attempting to access it fails with a record-locked error. A subsequent I/O operation on the logical unit unlocks the previously accessed record. Thus, no more than one record on a logical unit is ever locked.

Locked records can be explicitly unlocked by means of VAX FORTRAN's UNLOCK statement. The use of this statement minimizes the amount of time that a record is locked against access by other programs. The UNLOCK statement should be used in programs that retrieve records from a shared file but do not attempt to update them. See the *VAX FORTRAN Language Reference Manual* for additional information about the UNLOCK statement and its syntax.

For additional information about record locking for shared files, see the *Guide to VMS File Applications.*

See the section on condition handling in Chapter 14 for information on how to handle record locking for indexed files.

8.1.4 Using Mailboxes to Pass Information

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 you to pass data from one process to another. Mailboxes are created by the Create Mailbox system service (SYS$CREMBX). The following sections describe how to create mailboxes and how to send and receive data using mailboxes.

8.1.4.1 Creating a Mailbox

SYS$CREMBX creates the mailbox and returns the number of the I/O channel assigned to the mailbox. You must specify a variable for the I/O channel. You should also specify a logical name to be associated with the mailbox. The logical name identifies the mailbox for other processes and for VAX FORTRAN I/O statements. The SYS$CREMBX system service also allows you to specify the message and buffer sizes, the mailbox protection code, and the access mode of the mailbox; however, the default values for these arguments are usually sufficient.

The following segment of code creates a mailbox named MAILBOX. The number of the I/O channel assigned to the mailbox is returned in ICHAN.

```
INCLUDE '($SYSSRVNAM)'
INTEGER*2 ICHAN
ISTATUS = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
```

For more information about calling system services, see Chapter 6. For more information about the arguments supplied to the Create Mailbox system service, see the *VMS System Services Reference Manual.*

NOTE

Do not use MAIL as the logical name for a mailbox. If you do so, the system will not execute the proper image in response to the VMS command MAIL.

8.1.5 Sending and Receiving Data Using Mailboxes

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

Use VAX FORTRAN formatted sequential READ and WRITE statements to send and receive messages. The data transmission is performed synchronously. That is, a program that writes a message to a mailbox waits until the message is read, and a program that reads a message from a mailbox waits until the message is written before it continues transmission. When the writing program closes the mailbox, an end-of-file condition is returned to the reading program.

Do not attempt to write a record of zero length to a mailbox; the program reading the mailbox interprets this record as an end-of-file. Zero-length records are produced by consecutive slashes in FORMAT statements.

The sample program below creates a mailbox assigned with the logical name MAILBOX. The program then performs an open operation specifying the logical name MAILBOX as the file to be opened. It then reads file names from FNAMES.DAT and writes them to the mailbox until all of the records in the file have been transmitted.

```
      CHARACTER FILENAME*64
      INCLUDE '($SYSSRVNAM)'
      INTEGER*2 ICHAN
      INTEGER*4 STATUS

      STATUS = SYS$CREMBX(,ICHAN,,,,,'MAILBOX')
      IF (.NOT. STATUS) GO TO 99

      OPEN (UNIT=9, FILE='MAILBOX',
     1      STATUS='NEW', CARRIAGECONTROL='LIST', ERR=99)
      OPEN (UNIT=8, FILE='FNAMES.DAT', STATUS='OLD')

10    READ (8,100,END=98) FILENAME
      WRITE (9,100) FILENAME

100   FORMAT(A)
      GO TO 10

98    CLOSE (UNIT=8)
      CLOSE (UNIT=9)
      STOP

99    WRITE (6,*) 'Mailbox error'
      STOP
      END
```

The sample program below reads messages from a mailbox that was assigned the logical name MAILBOX when it was created. The messages comprise file names, which the program reads. The program then types the files associated with the file names.

```
      CHARACTER FILNAM*64, TEXT*133
      OPEN (UNIT=1, FILE='MAILBOX', STATUS='OLD')
1     READ (1,100,END=12) FILNAM
100   FORMAT (A)
      OPEN (UNIT=2, FILE=FILNAM, STATUS='OLD')
      OPEN (UNIT=3, FILE='SYS$OUTPUT', STATUS='NEW')

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
```

8.2 Remote Processes—Sharing and Exchanging Data

If your computer is a node in a DECnet–VAX network, you can communicate with other nodes in the network by means of standard VAX FORTRAN I/O statements. These statements let you exchange data with a program at the remote computer (task-to-task communication) and access files at the remote computer (resource sharing). There is no apparent difference between these intersystem exchanges and the local interprocess and file access exchanges.

Remote file access and task-to-task communications are discussed separately in the sections that follow.

8.2.1 Remote File Access

To access a file on a remote system, include the remote node name in the file name specification. For example:

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

To make a program independent of the physical location of the files it accesses, you can assign a logical name to the network file specification as shown in the following example:

```
$ ASSIGN MIAMI::DR4:[INV]INVENT.DAT INVFILE
```

The logical name INVFILE now refers to the remote file and can be used in the program. For example:

```
OPEN (UNIT=10, FILE='INVFILE', STATUS='OLD')
```

To process a file on the local network node, reassign the logical name; you do not need to modify the source program.

8.2.2 Network Task-to-Task Communication

Network task-to-task communication allows a program running on one network node to interact with a program running on another network node. This interaction is accomplished with standard VAX FORTRAN I/O statements and looks much like an interactive program/user session.

The steps involved in network task-to-task communications are as follows:

1. **Request the network connection.** The originating program initiates task-to-task communication. It opens the remote task file with a special file name syntax: the name of the remote task file is preceded with TASK= and surrounded with quotation marks. For example:

   ```
   BOSTON::"TASK=UPDATE"
   ```

   Unless the remote task file is contained in the default directory for the remote node's DECnet account, you must specify the pertinent account information (a user name and password) as part of the node name:

   ```
     BOSTON"username password"::"TASK=UPDATE"
   ```

   The form of the remote task file varies, depending on the remote computer's operating system. For VMS systems, this task file is a command file with a file type of COM. The network software submits the command file as a batch job on the remote system.

2. **Complete the network connection.** When the remote task starts, it must complete the connection back to the host. On VMS, the remote task completes this connection by performing an open operation on the logical name SYS$NET. When opening the remote task file or SYS$NET, specify either FORM='UNFORMATTED' or the combination of FORM='FORMATTED' and CARRIAGECONTROL='NONE'.

3. **Exchange messages.** When the connection is made between the two tasks, each program performs I/O using the established link.

   Task-to-task communication is synchronous. This means that when one task performs a read, it waits until the other task performs a write before it continues processing.

4. **Terminate the network connection.** To prevent losing data, the program that receives the last message should terminate the network connection using the CLOSE statement. When the network connection is terminated, the cooperating image receives an end-of-file error.

The following is a complete example showing how VAX FORTRAN programs can exchange information over a network. In this example, the originating program prompts for an integer value and sends the value to the remote program. The remote program then adds one to the value and returns the value to the originating program. It is assumed that the remote operating system is a VMS system.

The originating program on the local node contains the following source code:

```
      OPEN (UNIT=10, FILE='PARIS::"TASK=REMOTE"',
      1     STATUS='OLD', FORM='UNFORMATTED',
      2     ACCESS='SEQUENTIAL', IOSTAT=IOS, ERR=9999)
C Prompt for a number

      PRINT 101
101   FORMAT ($,' ENTER A NUMBER: ')
      ACCEPT *,N
C  Perform the network I/O

      WRITE (UNIT=10, IOSTAT=IOS, ERR=9999) N
      READ (UNIT=10, IOSTAT=IOS, ERR=9999) N

C Output the number and process errors

      PRINT 102, N
102   FORMAT (' The new value is ',I11)
      GO TO 99999
9999  PRINT *, 'Unexpected I/O Error Number ', IOS
99999 CLOSE (UNIT=10)
      END
```

The task file REMOTE.COM on the remote node contains the following VMS commands:

```
$ DEFINE SYS$PRINT NL:              ! Inhibit printing of log
$ RUN DBO:[NET]REMOTE.EXE           ! Run remote program
$ PURGE/KEEP=2 REMOTE.LOG           ! Delete old log files
```

The remote program PARIS::DB0:[NET]REMOTE.EXE contains the following source code:

```
OPEN (UNIT=10, FILE='SYS$NET', FORM='UNFORMATTED',
1     ACCESS='SEQUENTIAL', STATUS='OLD')
READ (UNIT=10) N
N = N + 1
WRITE (UNIT=10) N
CLOSE (UNIT=10)
END
```

For more information on using DECnet, refer to the *VMS Networking Manual* and *Introduction to DECnet.*

Chapter 9

Condition-Handling Facilities

An *exception condition,* as the term is used in this chapter, is an event, usually an error, that occurs during the execution of a program and is detected by system hardware or software or by logic in a user application program. A special type of routine, known as a *condition-handler routine,* is used to resolve exception conditions.

This chapter does not address error handling in a general sense, but only as it relates to the creation and use of condition-handler routines. (Refer to Chapter 5 for a general discussion of error handling.)

Examples of the types of exception conditions detected by system hardware and software are as follows:

- Hardware exceptions include such things as floating overflows, memory access violations, and the use of reserved operands.
- Software exceptions include such things as output conversion errors, end-of-file conditions, and invalid arguments to mathematical procedures.

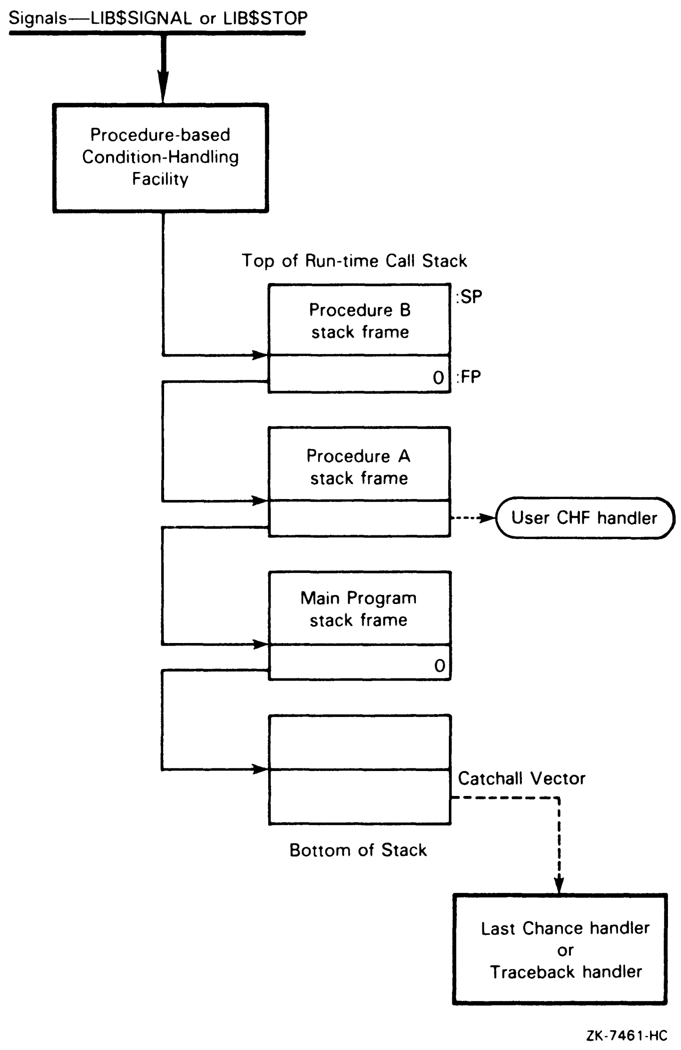
When an exception condition is detected by system hardware or software or by your program, that condition is *signaled* (by means of a *signal call*) to the condition-handling facility (CHF). The CHF then invokes one or more condition-handler routines that will attempt to either resolve the condition or terminate the processing in an orderly fashion.

The CHF allows a main program and each subprogram that follows it, regardless of call depth, to establish a condition-handler routine (one per program unit). Each of these condition-handler routines can potentially handle any or all software or hardware events that are treated as exception conditions by the user program or by the system hardware or software. Note that more than one condition handler for a given condition may be established by different program units in the call stack.

The address of the condition handler for a particular program unit is placed in the call frame for that unit in the run-time call stack. (The presence of a condition handler is indicated by a nonzero address in the first longword of the program unit's stack frame.)

Figure 9–1 shows the run-time call stack.

Figure 9–1: Sample Stack Scan for Condition Handlers


Signals—LIB$SIGNAL or LIB$STOP
Procedure-based Condition-Handling Facility
Top of Run-time Call Stack
Procedure B stack frame
:SP
0
:FP
Procedure A stack frame
User CHF handler
Main Program stack frame
0
Catchall Vector
Bottom of Stack
Last Chance handler or Traceback handler
ZK-7461-HC

Notes to Figure 9–1

1. The lines to the left of the run-time call stack depict the search path taken by the CHF, not pointers or control flow.
2. User CHF handlers are strictly optional and can be established for any of the program units (procedures) in an application program. For example, handlers for procedure B and the main program could have been shown in this diagram—in addition to the handler actually shown for procedure A.

When the program unit returns to its caller, the call frame is removed and the condition handler for that program unit can no longer be accessed by the CHF. (Multiple condition handlers may be accessed by the CHF in the processing of a single exception condition signal. This is discussed in Section 9.1.3.3.)

Throughout this chapter, the term *program unit* refers to an executable FORTRAN main program, subroutine, or function.

The remainder of this chapter describes the Condition-Handling Facility (CHF) in detail—how it operates, how user programs can interact with it, and how users can code their own condition-handling routines.

9.1 Using the Condition-Handling Facility

The Condition-Handling Facility (CHF) receives control and coordinates processing of all exception conditions that are signaled to it. The signals are issued under the following circumstances:

- When a user program detects an application-dependent exception condition
- When a VAX FORTRAN system hardware or software component detects a system-defined exception condition

In cases where the default condition handling is insufficient (see Section 9.1.1), you can develop your own handler routines and use the routine LIB$ESTABLISH to identify your handlers to the CHF. Typically, your needs for special condition handling are limited to the following types of operations:

- To respond to condition codes that are signaled instead of being returned, as in the case of integer overflow errors. (Section 9.1.4.6 describes the system-defined handler LIB$SIG_TO_RET. It allows you to treat signals as return values.)

- To modify part of a condition code, for example, the severity (see Section 9.1.2.3). (If you want to change the severity of any condition code to a severe error, you can optionally issue a call to LIB$STOP instead of writing a condition handler for that purpose.)
- To add additional messages to those messages associated with the originally signaled condition code or to log the occurrence of various application-specific or system-specific conditions.

When an exception condition is detected by a system hardware or software component or by a component in the user application program, the component calls the CHF by means of a signal routine (LIB$SIGNAL or LIB$STOP), passing a value to the CHF that identifies the condition. The CHF takes program control away from the routine that is currently executing and begins searching for a condition-handler routine to call. If it finds one, it establishes a call frame on the run-time call stack and then invokes the handler. The handler routine then attempts to deal with the condition.

The sections that follow describe the CHF in detail—how it operates, how user programs can interact with it, and how users can code their own condition-handling routines.

- Section 9.1.1 describes default condition handlers established by the system.
- Section 9.1.2 describes how a user program makes a condition handler known to the CHF and how it signals a condition and passes arguments.
- Section 9.1.3 describes how to write a condition-handling routine.
- Section 9.1.4 describes several condition-handling routines available in the VAX FORTRAN Run-Time Library.
- Section 9.1.5 contains some examples of the use of condition handlers.

9.1.1 Default Condition Handler

When the system creates a VAX FORTRAN user process, it establishes a system-defined condition handler that will be invoked by the CHF in the following circumstances:

- No user-established condition handlers exist in the call stack. (Any user-established condition handlers in the call stack are always invoked before the default handler is invoked.)
- All of the user-established condition handlers in the call stack return the condition code SS$_RESIGNAL to the CHF. (The SS$_RESIGNAL condition code causes the CHF to search for another condition handler. See Section 9.1.3.3.)

When establishing the default handler, the system has two handlers to chose from: the traceback handler and the catchall handler.

- *Traceback Handler.* Displays the message associated with the signaled condition code, the traceback message, the program unit name and line number of the statement that resulted in the exception condition, and the relative and absolute program counter values. In addition, the traceback handler displays the names of the program units in the current calling sequence and the line number of the invocation statements. (For exception conditions with a severity level of warning or error, the number of the next statement to be executed is also displayed.)

  After displaying the error information, the traceback handler continues program execution or, if the error is severe, terminates program execution. If the program terminates, the condition value (see Table 5–1) becomes the program exit status.
- *Catchall Handler.* Displays the message associated with the condition code and then either continues program execution or, if the error is severe, terminates execution. If the program terminates, the condition value (see Table 5–1) becomes the program exit status.

The /DEBUG and /TRACEBACK qualifiers—on the FORTRAN and LINK command lines, respectively—determine which default handler is enabled. If you take the defaults for these qualifiers, the traceback handler is established as the default handler. To establish the catchall handler as the default, you would specify /NODEBUG or /DEBUG=NOTRACEBACK on the FORTRAN command line and /NOTRACEBACK on the LINK command line.

9.1.2 User-Program Interactions with the CHF

User-program interactions with the CHF are strictly optional and application dependent. In each program unit, you have the option of establishing (and removing) a single condition handler to handle exceptions that may occur in that program unit or in subsequent subprograms (regardless of call depth). Once a program unit returns to its caller, its call frame is removed and any condition handler that the program unit has established thus becomes inaccessible.

The condition handler established by the user program can be coded to handle an exception condition signaled by either system hardware, a VAX FORTRAN system software component, or the user program itself. User-program signals are issued by means of the LIB$STOP and LIB$SIGNAL routines described in Section 9.1.2.2.

Although condition handlers offer a convenient and structured approach to handling exception conditions, they can have a significant impact on run-time performance. For commonly occurring application-specific conditions within a loop, for example, it may be wise to use other methods of dealing with the conditions. The best use of the facility is in large applications in which occasional exception conditions requiring special handling are anticipated.

The following sections describe how to establish and remove condition handlers and how to signal exception conditions.

9.1.2.1 Establishing and Removing Condition Handlers

Establishing a condition handler involves placing the address of a condition handling routine in the stack frame (longword 0) for the current program unit in the run-time call stack. This is done by issuing a call to the LIB$ESTABLISH routine. The form of the call is as follows:

```
CALL LIB$ESTABLISH (new-handler)
```

new-handler

Is the name of the routine to be set up as a condition handler.

LIB$ESTABLISH moves the address of the condition-handling routine into longword 0 of the calling program unit's stack frame and returns the previous contents of longword 0.

When the CHF receives control as a result of an exception condition, it searches for condition handlers—beginning with the stack frame for the current program unit. If the current program unit has not established a condition handler, the CHF searches the previous stack frames for a handler until it either finds a user-established condition handler or reaches the default condition handler established by the system.

The handler itself could be user written or selected from a list of utility functions provided with VAX FORTRAN. The following example shows how a call to establish a user-written handler might be coded.

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

In the preceding example, HANDLER is the name of a FORTRAN function subprogram that is established as the condition handler for the program unit containing these source statements. A program unit can remove an established condition handler in two ways:

- Issue another LIB$ESTABLISH call specifying a different handler
- Issue the LIB$REVERT call

The LIB$REVERT call has no arguments and is issued as follows:

```
CALL LIB$REVERT
```

This call removes the condition handler established in the current program unit.

When the program unit returns to its caller, the condition handler associated with that program unit is automatically removed (that is, the program unit's stack frame, which contains the condition handler address, is removed from the stack).

9.1.2.2 Signaling a Condition

When a prescribed condition requiring special handling by a condition handler is detected by logic in your program, you issue a condition signal in your program in order to invoke the CHF. A condition signal consists of a call to one of the two system-supplied signal routines in either of the following forms:

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

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

condition-value
Is an INTEGER*4 value that identifies a particular exception condition (see Section 9.1.2.3), and can only be passed using the %VAL argument-passing mechanism.

arg
Are optional arguments to be passed to user-established condition handlers and the system default condition handlers. These arguments consist of messages and formatted-ASCII-output arguments (see the *VMS Run-Time Library Routines Volume*).

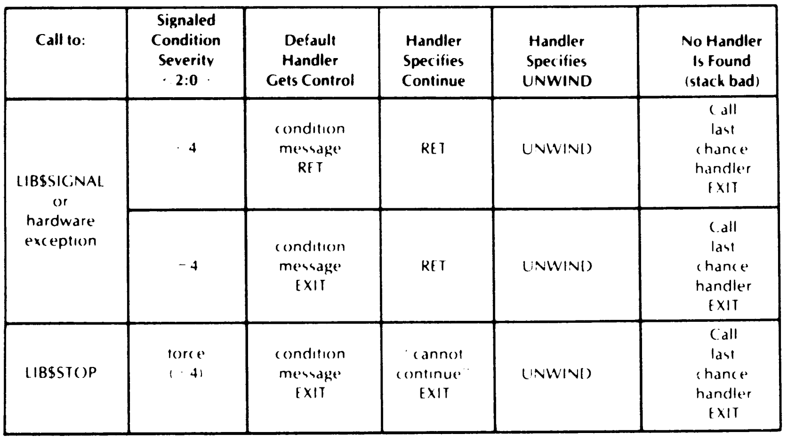
The CHF uses these parameters to build the signal argument array SIGARGS (see Section 9.1.3.2) before passing control to a condition handler.

Whether you issue a call to LIB$SIGNAL or LIB$STOP depends on the following considerations:

- If the current program unit can continue after the signal is made, call LIB$SIGNAL. The condition handler can then determine whether program execution continues. After the signal is issued, control is never returned to the user program until one of the condition handlers in the call stack resolves the exception condition and indicates to the CHF that program execution should continue.
- If the condition does not allow the current program unit to continue, call LIB$STOP. Note that the only way to override a LIB$STOP signal is to perform an *unwind* operation (see Section 9.1.3.3).)

Table 9–1 lists all of the possible effects of a LIB$SIGNAL or LIB$STOP call.

Table 9–1: Effects of Calls to LIB$SIGNAL or LIB$STOP



Call to:	Signaled Condition Severity <2:0>	Default Handler Gets Control	Handler Specifies Continue	Handler Specifies UNWIND	No Handler Is Found (stack bad)
LIB$SIGNAL or hardware exception	< 4	condition message RET	RET	UNWIND	Call last chance handler EXIT
	= 4	condition message EXIT	RET	UNWIND	Call last chance handler EXIT
LIB$STOP	force (= 4)	condition message EXIT	"cannot continue" EXIT	UNWIND	Call last chance handler EXIT

ZK-5162-86

In Table 9–1, "cannot continue" indicates an error that results in the following message:

```
IMPROPERLY HANDLED CONDITION, ATTEMPT TO CONTINUE FROM STOP
```

To pass the condition value, you must use the %VAL argument-passing mechanism (see Section 6.2.3). Condition values are usually expressed as condition symbols (see Section 6.5.1). Condition symbols have the following forms:

```
    fac$_symbol (DIGITAL-defined)
```

or

```
    fac__symbol (user-defined)
```

fac
Is a facility name prefix.

symbol
Identifies a specific condition. (Refer to Table 5–1 for a list of FORTRAN-related condition symbols.)

In the following example, a signal call passes a condition symbol associated with the mathematical procedures library.

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

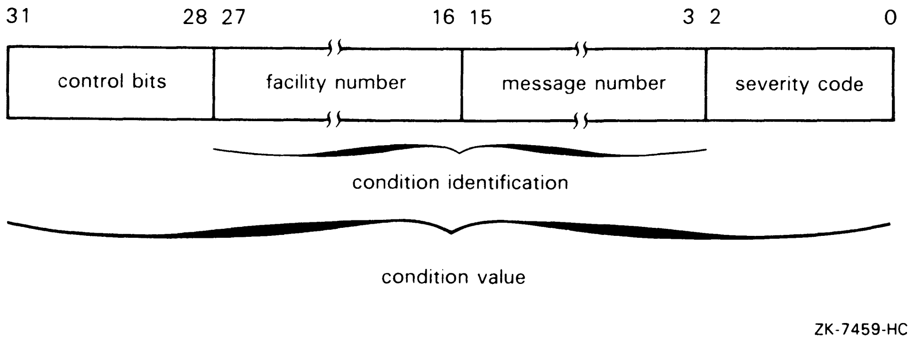
You can include additional arguments to provide supplementary information about the error.

When your program issues a condition signal, the CHF searches for a condition handler by examining the preceding call frames, in order, until it either finds a procedure that handles the signaled condition or reaches the default condition handler.

The following section describes condition values and condition symbols in detail.

9.1.2.3 Condition Values and Symbols Passed to CHF

The VMS system uses condition values to indicate that a called procedure has either executed successfully or failed, and to report exception conditions. Condition values are INTEGER*4 values. They consist of fields that indicate which software component generated the value, the reason the value was generated, and the severity of the condition. A condition value has the following fields:



The facility number field identifies the software component that generated the condition value. Bit 27 = 1 indicates a user-supplied facility; bit 27 = 0 indicates a system facility.

The message number field identifies the condition that occurred. Bit 15 = 1 indicates that the message is specific to a single facility; bit 15 = 0 indicates a system-wide message.

Table 9-2 gives the meanings of values in the severity code field.

Table 9–2: Severity Codes for Exception Condition Values

Code (Symbolic Name)	Severity	Response
0 (STS$K_WARNING)	Warning	Execution continues, unpredictable results
1 (STS$K_SUCCESS)	Success	Execution continues, expected results
2 (STS$K_ERROR)	Error	Execution continues, erroneous results
3 (STS$K_INFORMATION)	Information	Execution continues, informational message displayed
4 (STS$K_SEVERE)	Severe error	Execution terminates, no output
5 - 7	–	Reserved for use by DIGITAL

The symbolic names for the severity codes are defined in the $SSDEF module in the FORTRAN Symbolic Definition Library (FORSYSDEF).

A condition handler can alter the severity code of a condition value—either to allow execution to continue or to force an exit, depending on the circumstances.

The condition value is passed in the second element of the array SIGARGS. (See Section 9.1.3.2 for detailed information about the contents and use of the array SIGARGS.) In some cases, 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, only if a previous handler has not changed the severity code).

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 routine (see Section 9.1.4.4).

The FORTRAN Symbolic Definition Library (FORSYSDEF) contains modules that define condition symbols. When you write a condition handler, you can specify any of the following modules, as appropriate, with an INCLUDE statement:

- **$FORDEF**—This module contains definitions for all condition symbols from the FORTRAN-specific library routines. Refer to Table 5–1 for a list of the FORTRAN error numbers (IOSTAT values) associated with these symbols. These symbols have the form:

  ```
     FOR$_error
  ```

  For example:

  ```
  FOR$_INPCONERR
  ```

- **$LIBDEF**—This module contains definitions for all condition symbols from the VMS general utility library facility. These symbols have the form:

  ```
     LIB$_condition
  ```

  For example:

  ```
  LIB$_INSVIRMEM
  ```

- **$MTHDEF**—This module contains definitions for all condition symbols from the mathematical procedures library. These symbols have the form:

  ```
     MTH$_condition
  ```

  For example:

  ```
  MTH$_SQUROONEG
  ```

- **$SSDEF**—This module contains definitions for system services status codes, which are frequently used in FORTRAN condition handlers. These symbols have the form:

  ```
     SS$_status
  ```

  For example:

  ```
  SS$_FLTOVF
  ```

9.1.3 How to Write a Condition Handler

The following sections describe how to code condition handlers for your own applications.

9.1.3.1 Operations Performed in Handlers

A condition handler responds to an exception by analyzing arguments passed to it and by taking appropriate action. Possible actions taken by condition handlers are as follows:

- Condition correction
- Condition reporting
- Execution control

First, the handler must determine whether it can correct the condition identified by the condition code passed by the signal call. If possible, the handler takes the appropriate corrective action and execution continues. If it cannot correct the condition, the handler may resignal the condition. That is, it may request that another condition handler, associated with an earlier program unit in the call stack, attempt to process the exception.

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

- Maintaining a count of exceptions encountered during program execution.
- Signaling the same condition again (that is, resignaling) in order 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 appropriate to a specific application. (The condition handler must establish the application-specific condition handler using LIB$ESTABLISH and then signal the condition using LIB$SIGNAL.)

Execution can be affected in a number of ways, such as:

- Continuing from the point of exception. However, if the signal was issued by means of a call to LIB$STOP, the program exits.
- Returning control (unwinding) to the program unit that established the handler. Execution resumes at the point of the call that resulted in the exception. The handler establishes the function value to be returned by the called procedure.
- Returning control (unwinding) to the establisher's caller (that is, to the program unit that called the program unit that established the handler). The handler establishes the function value to be returned by the program unit that established the handler.

See Section 9.1.3.3 for information about returning from condition handlers.

9.1.3.2 Coding Requirements of Condition Handlers

A VAX FORTRAN condition handler is an INTEGER*4 function that has two argument arrays passed to it by the CHF. To meet these requirements, you could define a condition handler as follows:

```
INTEGER*4  FUNCTION HANDLER(SIGARGS,MECHARGS)
INTEGER*4  SIGARGS(*), MECHARGS(5)
```

The CHF creates the signal and mechanism argument arrays SIGARGS and MECHARGS and passes them to the condition handler.

The array SIGARGS is used by condition handlers to obtain information passed as arguments in the LIB$SIGNAL or LIB$STOP signal call. The contents of SIGARGS are as follows:

Array Element	**Contents**
SIGARGS(1)	Argument count
SIGARGS(2)	Condition code
SIGARGS(3 to *n*-1)	Zero or more additional arguments
.	
.	
SIGARGS(*n*)	PC (program counter)
SIGARGS(*n*+1)	PSL (processor status longword)

- The first array element, SIGARGS(1), indicates how many additional arguments are being passed in this array. The count does not include this first element.
- The second element, SIGARGS(2), indicates the signaled condition (condition value) specified by the call to LIB$SIGNAL or LIB$STOP. If more than one message is associated with the exception condition, the condition value in SIGARGS(2) belongs to the first message. See Section 9.1.2.3 for a discussion of condition values.
- The third element, SIGARGS(3), is the message description for the message associated with the condition code in SIGARGS(2). The format of the message description varies depending on the type of message being signaled. For more information, see the SYS$PUTMSG description in the *VMS System Services Reference Manual.*

  Additional arguments, SIGARGS(*n*-1), may be specified in the call to LIB$SIGNAL or LIB$STOP (see Section 9.1.2.2).
- The second-to-last element, SIGARGS(*n*), contains the value of the program counter (PC).

  If the condition that caused the signal was a fault (occurring during the instruction's execution), the PC contains the address of the call instruction that signaled the condition code.

  If the condition that caused the signal was a trap (occurring at the end of the instruction), the PC contains the address of the instruction following the call that signaled the condition code.

  See Section 9.1.4.5 for additional information about faults and traps.
- The last element, SIGARGS(*n*+1), reflects the value of the processor status longword (PSL) at the time the signal was issued.

A condition handler is usually written in anticipation of a particular condition code or set of condition codes. Because handlers are invoked as a result of any signaled condition code, you should begin your handler routine by comparing the condition code passed to the handler (element 2 of SIGARGS) against the condition codes expected by the handler. If the signaled condition code is not an expected code, you should resignal the condition code by equating the function value of the handler to the global symbol SS$_RESIGNAL (see Section 9.1.3.3).

The array MECHARGS is used to obtain information about the procedure activation of the program unit that established the condition handler. MECHARGS is a 5-element array; its values are defined as follows:

Array Element	Contents
MECHARGS(1)	Argument count
MECHARGS(2)	Establisher
MECHARGS(3)	Call depth
MECHARGS(4)	Function value (R0)
MECHARGS(5)	R1

- MECHARGS(1) contains the argument count of this array, not including this first element (that is, the value 4).
- MECHARGS(2) contains the address of the call frame for the program unit that established the handler.
- MECHARGS(3) contains the number of calls made between the program unit that established the handler and the program unit that signaled the condition code.
- MECHARGS(4) and MECHARGS(5) contain the values of registers R0 and R1 at the time of the signal. When execution continues or when a stack unwind occurs, these values are restored to R0 and R1.

  By changing these register values, a handler performing an unwind can alter the function value returned to a program unit (see Section 9.1.3.3 for details about an unwind operation).

Inside a condition handler, you can use any other variables that you need to use. If they are shared with other program units (for example, in common blocks), make sure that they are declared volatile. This will ensure that compiler optimizations do not invalidate the handler actions. See Section 11.3.2.2 and the *VAX FORTRAN Language Reference Manual* for more information on the VOLATILE statement.

9.1.3.3 Returning from a Condition Handler

One way that condition handlers control subsequent execution is by specifying a function return value. Function return values and their effects are defined in Table 9–3.

Table 9–3: Condition-Handler Function Return Values

Symbolic Values	Effects
SS$_CONTINUE	If you equate the function value of the condition handler to SS$_CONTINUE, the handler returns control to the program unit at the statement that signaled the condition (fault) or the statement following the one that signaled the condition (trap). (The effects of faults and traps are described in Section 9.1.4.5.)
SS$_RESIGNAL	If you equate the function value of the condition handler to SS$_RESIGNAL or do not specify a function value (function value of zero), the CHF will search for another condition handler in the call stack. If you modify SIGARGS or MECHARGS before resignaling, the modified arrays are passed to the next handler.

Alternatively, a condition handler can request a call stack unwind by calling SYS$UNWIND before returning. Unwinding the call stack removes call frames, starting with the call frame for the program unit in which the condition occurred, and returns control to an earlier program unit in the current calling sequence. In this case, any function return values established by condition handlers are ignored by the CHF.

You can unwind the call stack whether the condition was detected by hardware or signaled by means of LIB$SIGNAL or LIB$STOP. Unwinding is the only way to continue execution after a call to LIB$STOP.

A stack unwind is typically made to one of two places:

- To the establisher of the condition handler that issues the SYS$UNWIND. To do this, you pass the call depth (third element of the MECHARGS array) as the first argument in the call to SYS$UNWIND. Do not specify a second argument. For example:

  ```
  CALL SYS$UNWIND(MECHARGS(3),)
  ```

  Control returns to the establisher and execution resumes at the point of the call that resulted in the exception.

- To the establisher's caller. To do this, do not specify any arguments in the call to SYS$UNWIND. For example:

  ```
  CALL SYS$UNWIND(,)
  ```

  Control returns to the program unit that called the establisher of the condition handler that issues the call to SYS$UNWIND.

The actual stack unwind is not performed immediately after the condition handler issues the call to SYS$UNWIND. It occurs when a condition handler returns control to the CHF.

During the actual unwinding of the call stack, SYS$UNWIND examines each frame in the call stack to determine whether a condition handler was declared. If a handler was declared, SYS$UNWIND calls the handler with the condition value SS_$UNWIND (indicating that the stack is being unwound) in the condition name argument of the signal array. When a condition handler is called with this condition value, that handler can perform any procedure-specific clean-up operations that may be required. After the condition handler returns, the call frame is removed form the stack.

Section 9.1.5 contains an example of the use of SYS$UNWIND.

9.1.4 Use of LIB$ Routines as Condition Handlers

In addition to the routines described previously in this section (LIB$ESTABLISH, LIB$REVERT, LIB$SIGNAL, and LIB$STOP), the Run-Time Library contains the following routines for use by condition handling routines or for use as condition handlers:

- Routines to enable or disable signaling of hardware exceptions:

LIB$DEC_OVER	Enables or disables signaling of decimal overflow
LIB$FLT_UNDER	Enables or disables signaling of floating-point underflow
LIB$INT_OVER	Enables or disables signaling of integer overflow

- Routines for use as condition handling routines:

LIB$FIXUP_FLT	Changes floating-point reserved operand to a specified value
LIB$MATCH_COND	Matches condition value
LIB$SIG_TO_RET	Converts any signal to return status
LIB$SIG_TO_STOP	Converts a signaled condition to a condition that cannot be stopped
LIB$SIM_TRAP	Simulates a floating-point trap

The sections that follow give details on the preceding routines.

9.1.4.1 Overflow/Underflow Detection Enabling Routines

You can use the following VAX FORTRAN Run-Time Library routines to enable or disable the signaling of decimal overflow, floating-point underflow, and integer overflow:

- LIB$DEC_OVER enables or disables the reporting of decimal overflow.
- LIB$FLT_UNDER enables or disables the reporting of floating-point underflow.
- LIB$INT_OVER enables or disables the reporting of integer overflow.

You cannot disable the signaling of integer divide-by-zero, floating-point overflow, and floating-point or decimal divide-by-zero.

When the signaling of a hardware condition is enabled, the occurrence of the exception condition results in a severe error. When the signaling of a hardware condition is disabled, the occurrence of the condition is ignored and the processor executes the next instruction in the sequence.

Each of the LIBDEC_OVER, LIBFLT_UNDER, and LIB$INT_OVER routines takes a single argument. The argument passed is the address of a BYTE value containing the setting you want to establish for decimal overflow, floating-point underflow, or integer overflow, respectively. Bit 0 set to 1 means enable; bit 0 set to 0 means disable.

Options relating to the processing of floating-point underflow exceptions are described at length in the next section.

9.1.4.2 Floating Underflow Exceptions

VAX FORTRAN, by default, does not enable underflow exceptions. If the result of an operation is smaller than the smallest representable floating-point number, the result is set to zero, program execution continues, and no error message is given.

VAX FORTRAN does, however, provide the facilities for reporting and processing underflow exceptions. You can enable underflow exceptions for all or part of a routine, and you can either choose VAX FORTRAN default processing or provide a user-written handler. These options are discussed under the headings that follow:

Specifying CHECK=UNDERFLOW

When you specify the /CHECK=UNDERFLOW option on either the FORTRAN command line or an OPTIONS statement, the compiler takes two actions:

- For subprograms and main programs, it generates code to enable underflow exceptions at the beginning of each routine.
- For main programs only, it causes the FORTRAN-specific condition handler FOR$UNDERFLOW_HANDLER to become established at run time in a call frame preceding the main program's call frame.

If an operation in the main program or subprogram underflows and it is not processed by another handler, FOR$UNDERFLOW_HANDLER assumes control and performs the following actions:

1. Increments a count of the number of underflows
2. Changes an underflow fault to an underflow trap
3. Prints an error message and generates a traceback for the first two underflows
4. Stores a zero in the result
5. Continues program execution

When the program exits, an informational message is printed, giving the total number of underflows generated by the program.

If the main program is not written in FORTRAN or if it is not compiled with the /CHECK=UNDERFLOW option, a system default handler, not FOR$UNDERFLOW_HANDLER, assumes control when an underflow exception occurs. It prints a traceback listing and terminates the program execution.

Establishing a Handler for Underflow

If you wish to handle underflow in a manner different from FOR$UNDERFLOW_HANDLER, you can establish a user handler (see Section 9.1.4.5). The user-established handler assumes control before the default handler (because the default handler is established in a frame above the main program). The user program thus has complete control over how the exception is processed.

Note that when you write a handler for floating underflow, you must be careful to account for both faults and traps (refer to Section 9.1.4.5).

In some cases, it is not possible to compile the main program with the /CHECK=UNDERFLOW option, for example, when the main program is not written in FORTRAN. Given these circumstances, if you would like to have the behavior of FOR$UNDERFLOW_HANDLER in effect for underflow exceptions, you could use the following code in any FORTRAN routine in your application program to establish FOR$UNDERFLOW_HANDLER explicitly:

```
EXTERNAL FOR$UNDERFLOW_HANDLER
CALL LIB$ESTABLISH(FOR$UNDERFLOW_HANDLER)
```

Any combination of routines in a program can establish FOR$UNDERFLOW_HANDLER without interfering with the proper behavior of the handler. However, make sure that you establish the handler before the occurrence of any operations that could cause underflow.

Enabling and Disabling Underflow Exceptions

You can enable or disable underflow exceptions for parts of a routine by invoking the routine LIB$FLT_UNDER as a function. LIB$FLT_UNDER takes one argument: to enable underflow in the current routine, specify 1; to disable underflow in the current routine, specify 0.

9.1.4.3 Floating Reserved Operand Faults

Certain errors resulting from floating-point operations generate special values, called floating reserved operands. Errors that occur in math library procedures and the floating overflow or division-by-zero traps are primary examples of conditions that produce floating reserved operands.

A floating reserved operand has a sign bit of 1 and an exponent of 0. A floating reserved operand fault occurs when a VAX floating-point instruction retrieves a floating reserved operand. To continue program execution after this fault, you must provide a condition handler that changes the reserved operand to a nonreserved value and restarts the instruction. As with other faults, if you restart the instruction without first correcting the condition, you create an infinite loop of faults.

You can use the routine LIB$FIXUP_FLT to replace a reserved operand with a new value and restart the instruction. This procedure can be either established as a condition handler or called from a user-written condition handler:

- The following statements establish LIB$FIXUP_FLT as a condition handler:

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

  By default, the procedure replaces the reserved operand with a zero value and continues program execution.

- The following statements invoke LIB$FIXUP_FLT from a user-written condition handler:

  ```
  INTEGER*4 FUNCTION HANDLER(SIGARGS, MECHARGS)
  INTEGER*4 SIGARGS(*), MECHARGS(5)
  HANDLER = LIB$FIXUP_FLT(SIGARGS, MECHARGS, 1.0E0)
  RETURN
  END
  ```

  This user-written condition handler returns the success or failure status value returned by LIB$FIXUP_FLT. If the correction is successful, execution continues; if not, the condition is resignaled. The third argument to LIB$FIXUP_FLT is optional and explicitly specifies the reserved operand replacement value; it must be a REAL*4 value. If you omit this argument, however, LIB$FIXUP_FLT automatically supplies a value of zero. (See Section 10.5.1.1 for additional information on floating-point reserved operands.)

9.1.4.4 Matching Condition Values to Determine Program Behavior

In many condition-handling situations, you may want to respond to an exception condition regardless of the value of the severity code passed in the condition value. To ignore the severity and control fields of a condition value, use the LIB$MATCH_COND routine as a function in the following form:

```
index = LIB$MATCH_COND(SIGARGS(2),con-1,...con-n)
```

index
Is an integer variable that is assigned a value for use in a subsequent computed GOTO statement.

con
Is a condition value.

The LIB$MATCH_COND function compares bits 27:3 of the value in SIGARGS(2) with bits 27:3 of each specified condition value. If it finds a match, the function assigns the index value 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 '($FORDEF)'
INTEGER*4 SIGARGS(*), 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, then INDEX = 0, and control transfers to the next executable statement after the computed GOTO. A match with any of the values in the list transfers control to the corresponding statement in the GOTO list. Thus, if SIGARGS(2) matches FOR$_OPEFAI, control transfers to statement 400. Note the use of condition symbols to represent condition values. Refer to Table 5–1 for a list of the FORTRAN-related condition symbols and their meanings.

9.1.4.5 Converting Faults and Traps

You can have your program signal a condition by calling LIB$SIGNAL or LIB$STOP directly, as described in Section 9.1.2.2. However, most conditions are signaled on behalf of your program by the system hardware or software in response to a system event. These conditions are processed by the CHF and handled in the same way as calls to LIB$SIGNAL and LIB$STOP. This section describes some of the system events that signal conditions on behalf of your program.

If a VAX processor detects an error while executing a machine instruction, it can take one of two actions.

- The first action, called a fault, preserves the contents of registers and memory in a consistent state so that the instruction can be restarted.
- The second action, called a trap, completes the instruction, but with a predefined result. For example, if an integer overflow trap occurs, the result is the correct low-order part of the true value.

The action taken when an exception occurs depends upon the type of exception. For example, faults are taken for access violations and for detection of a floating reserved operand. Traps are taken for integer overflow and for integer divide-by-zero exceptions. However, when a floating overflow, floating underflow, or floating divide-by-zero exception occurs, the action taken depends upon which type of VAX processor is executing the instruction.

- Early versions of the VAX–11/780 processor trap when these errors occur. For floating overflow or divide-by-zero, a floating reserved operand is stored in the destination; for floating underflow, a zero is stored in the destination. (Note that most of these processors have been updated and now assume the error-handling behaviors of newer processors.)
- All other VAX processors fault on these exceptions, allowing the error to be corrected and the instruction restarted.

If a program that expects floating traps runs on a VAX processor that faults, execution may continue incorrectly. For example, if a condition handler merely causes execution to continue after a floating trap, a reserved operand is stored and the next instruction is executed. However, the same handler used on a processor that faults causes an infinite loop of faults because it restarts the erroneous instruction. Therefore, you should write floating-point exception handlers that take the appropriate actions for both faults and traps.

Separate sets of condition values are signaled by the processor for faults and traps. Exceptions and their condition code names are as follows:

Exception	Fault	Trap
Floating overflow	SS$_FLTOVF_F	SS$_FLTOVF
Floating underflow	SS$_FLTUND_F	SS$_FLTUND
Floating divide-by-zero	SS$_FLTDIV_F	SS$_FLTDIV

To convert a floating-point fault to a floating-point trap, you can use the LIB$SIM_TRAP routine either as a condition handler or as a called routine from a user-written handler.

The arguments passed to LIB$SIM_TRAP are as follows:

```
LIB$SIM_TRAP (sig-args,mch-args)
```

sig-args
Contains the address of the signal argument array (see Section 9.1.3.2).

mch-args
Contains the address of the mechanism argument array (see Section 9.1.3.2).

LIB$SIM_TRAP simulates a floating-point fault as if a floating-point trap had occurred and sets the PC to point to the instruction after the one that caused the exception condition. Thus, it enables your program to continue execution without resolving the original condition. LIB$SIM_TRAP intercepts only floating-point overflow, underflow, and divide-by-zero faults.

Note that the PDP-11 FORTRAN compatibility error-processing routines ERRSET and ERRTST implicitly enable LIB$SIM_TRAP so that faults are converted to traps.

9.1.4.6 Changing a Signal to a Return Status

When it is preferable to detect errors by signaling, but the calling procedure expects a returned status, LIB$SIG_TO_RET may be used by the procedure that signals. LIB$SIG_TO_RET is a condition handler that converts any signaled condition to a return status. The status is returned to the caller of the procedure that established LIB$SIG_TO_RET.

The arguments for LIB$SIG_TO_RET are the same as those passed to LIB$SIM_TRAP (see Section 9.1.4.5).

You can establish LIB$SIG_TO_RET as a condition handler by specifying it in a call to LIB$ESTABLISH. You can also establish it by calling it from a user-written condition handler. If LIB$SIG_TO_RET is called from a condition handler, the signaled condition is returned as a function value to the caller of the establisher of that handler when the handler returns to the CHF. When a signaled exception condition occurs, LIB$SIG_TO_RET procedure does the following:

- Places the signaled condition value in the image of R0 that is saved as part of the mechanism argument vector.
- Calls the unwind system service ($UNWIND) with the default arguments. After returning from LIB$SIG_TO_RET (when it is established as a condition handler) or after returning from the condition handler that called LIB$SIG_TO_RET (when LIB$SIG_TO_RET is called from within a condition handler), the stack is unwound to the caller of the procedure that established the handler.

Your calling procedure is then able to test R0, as if the called procedure had returned a status, and specify an error recovery action.

9.1.4.7 Changing a Signal to a Stop

The routine LIB$SIG_TO_STOP causes a signal to appear as though it had been signaled by a call to LIB$STOP. LIB$SIG_TO_STOP can be established as a condition handler or called from within a user-written condition handler.

The argument that you passed to LIB$STOP is a 4-byte condition value (see Section 9.1.2.3). The argument must be passed using the %VAL argument-passing mechanism.

When a signal is generated by LIB$STOP, the severity code is forced to *severe* and control cannot be returned to the procedure that signaled the condition.

9.1.5 Condition Handler Examples

The examples in this section demonstrate the use of condition handlers in typical FORTRAN procedures.

Example 1:

The following example uses a matrix inversion procedure, with the logical function name INVERT, to indicate the success or failure of the procedure. That is, if the matrix can be inverted, INVERT returns the .TRUE. logical value. If the matrix is singular, INVERT returns the .FALSE. logical value. During execution of the matrix inversion procedure, 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(*), MECHARGS(5)
      INCLUDE '($SSDEF)'
      HANDL = SS$_RESIGNAL           ! ASSUME RESIGNAL

      IF (LIB$MATCH_COND(SIGARGS(2),
     1    SS$_FLTOVF,SS$_FLTOVF_F,SS$_FLTDIV,SS$_FLTDIV_F) .NE. 0) THEN
         MECHARGS(4) = .FALSE.
         CALL  SYS$UNWIND(,)
      END IF

      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 signaled condition and therefore tests the condition value (SIGARGS(2)).

If the condition is floating overflow or floating division-by-zero, 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$UNWIND and returns; the condition handling facility then gets control and actually performs the unwind operation. Note that the function value from the user-written condition handler (HANDL = SS$_RESIGNAL) is ignored if SYS$UNWIND is called.

If the exception condition is not a floating overflow or division-by-zero, the condition handler returns a value of SS$_RESIGNAL, indicating that it is unable to deal directly with the condition. The immediately preceding procedure activation is then checked for a condition handler, continuing until an established condition handler or the default condition handler is reached.

Example 2:

This example of a condition handler processes the conditions MTH$_FLOOVEMAT and MTH$_FLOUNDMAT. The purpose of the condition handler is to modify the value returned by the math run-time library from the default value (floating reserved operand -0.0) to the largest representable floating-point value, 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(*), MECHARGS(5)
      INCLUDE '($SSDEF)'
      INCLUDE '($MTHDEF)'

      IF (SIGARGS(2) .EQ. MTH$_FLOOVEMAT) THEN
          MECHARGS(4) = 'FFFF7FFF'X
          MECHARGS(5) = 'FFFFFFFF'X
          HDLR = SS$_CONTINUE
      ELSE
          HDLR = SS$_RESIGNAL
      END IF
      RETURN
      END
```

When an exception condition occurs, HDLR is called, and it compares the condition value (SIGARGS(2)) with MTH$_FLOOVEMAT. If the condition is not MTH$_FLOOVEMAT, then SS$_RESIGNAL is returned, and the preceding procedure activations are searched for an established handler.

The recovery technique used in HDLR depends upon a particular coding convention used in mathematical procedures in the Run-Time Library. If an error is detected in a math library procedure, the following steps are performed:

1. A default function value is stored in registers R0 and R1. Typically, this is the floating reserved operand, -0.0.
2. An exception condition is signaled by calling LIB$SIGNAL. The contents of registers R0 and R1 are preserved in the mechanism vector (MECHARGS(4), MECHARGS(5)).
3. The value in registers R0 and R1 that exists following the signal is stored as the function value.

If the condition is MTH$_FLOOVEMAT, HDLR stores the largest representable floating-point value in the saved R0 and R1 elements of array MECHARGS. HDLR returns the function value SS$_CONTINUE, and execution continues in the math library procedure. No error message is printed. The values in MECHARGS(4) and MECHARGS(5) are restored to R0 and R1 by the condition handling facility; they are then returned as the function value by the math library procedure.

Example 3:

This example of a condition handler determines the reason for a system service failure. The condition handler only handles one type of exception, namely, system service failures. All other exceptions are resignaled, allowing them to be handled by the system default handlers. This condition handler is useful because the system traceback handler only indicates that a system service failure occurred, not which specific error caused the failure.

Source Program:

```
C                           SSCOND.FOR
C
C       This program defines and establishes its own
C       condition handling routine to handle system service
C       failures.
C
      IMPLICIT       INTEGER*4 (A-Z)
      EXTERNAL       SSHAND
C
C       Establish condition handler
C
      CALL LIB$ESTABLISH (SSHAND)                          ❶

      TYPE *, 'Handler established.'
C
C       Enable system service failure mode
C
      CALL SYS$SETSFM (%VAL(1))                            ❷
C
C       Generate a bad system service call
C

      CALL SYS$QIOW(,,,,,,,,,,,)                           ❸
C
      END
C
      INTEGER*4 FUNCTION SSHAND (SIGARGS, MECHARGS)        ❹
C
C       This routine is to be used as a condition handler
C       for system service failures.
C
      IMPLICIT       INTEGER*4 (A-Z)
      INTEGER*4      SIGARGS(*), MECHARGS(5)               ❺
      INTEGER*2      MSGLEN
      CHARACTER*120  ERRMSG
      INCLUDE        '($SSDEF)'
C
C       If not system service fail, resignal
C
      IF (SIGARGS(2) .NE. SS$_SSFAIL) THEN
             SSHAND= SS$_RESIGNAL                          ❻
C
C       If system service failure
C
      ELSE
             STAT=SYS$GETMSG( %VAL(SIGARGS(3)), MSGLEN,
     1                                 ERRMSG,,)           ❼
             IF (.NOT. STAT) CALL LIB$STOP(%VAL(STAT))
C
             TYPE *, 'System service call failed with error:'
             TYPE *, ERRMSG(1:MSGLEN)
C
C              This is where the handler would perform
```

```
C               corrective measures.
C
              SSHAND = SS$_RESIGNAL                              ❽
       ENDIF
C
       RETURN
       END
```

Program Output:

```
$RUN SSCOND
Handler established
System service call failed with error:                          ❾
%SYSTEM-F-IVCHAN, invalid I/O channel
%SYSTEM-F-SSFAIL, system service failure exception, status=0000013C,    ❿
PC=7FFEDE06, PSL=03C00000
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name     line     relative PC     absolute PC
                                          7FFEDE06        7FFEDE06
SSCOND$MAIN     SSCOND$MAIN        23     0000003B        0000063B
```

Notes to Example 3:

❶ LIB$ESTABLISH is used by the main program to establish the condition handler SSHAND.

❷ System service failure mode is enabled so errors in system service calls will initiate a search for a condition handler. The system service $SETSFM allows system service errors to be signaled. Therefore, the program need not check error status after each system service call. The condition handler can respond to all errors generated by system service calls.

❸ A system service error is generated by not specifying any arguments to $QIOW. The LIB$SIGNAL routine could also be used here to generate any exception condition name to test the condition handler.

❹ SSHAND is declared as an INTEGER*4 function in order to enable it to return a status code in R0.

❺ The signal and mechanism arrays must be dimensioned. Notice that the mechanism array always contains five elements, but the signal array varies according to the number of additional arguments.

❻ The handler checks the error condition to determine whether it is one of the conditions that it can handle. The LIB$MATCH_COND routine would be useful here if the routine wanted to check for one of a collection of conditions. The condition handler should always test for specific errors, and handle only those errors for which it is written. Other errors should simply be resignaled.

❼ The $GETMSG system service is used to translate the error code into the associated error message.

❽ If the routine did not remedy the exception condition, it will return with a value of SS$_RESIGNAL.

❾ Output from user-written condition handling routine.

❿ Output from the system-defined condition handlers.

Chapter 10

VAX FORTRAN Implementation Notes

This chapter discusses aspects of the relationship between the VAX FORTRAN language and its implementation on the VAX system. The purpose is to provide insights that will allow you to use VAX FORTRAN in a way that makes the best use of its features. The following topics are addressed:

- Program sections
- Storage allocation
- DO loops
- ENTRY statement arguments
- Floating-point data representation

VAX FORTRAN calling conventions are treated separately in Chapter 6.

10.1 VAX FORTRAN Program Section Usage

The storage required by a VAX FORTRAN program unit is allocated in contiguous areas called program sections (PSECTs). The VAX FORTRAN compiler implicitly declares three PSECTs:

$CODE	—	Contains all executable code.
$PDATA	—	Contains read-only data (for example, constants).
$LOCAL	—	Contains read/write data that is 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.) Memory allocation and sharing are controlled by the linker according to the attributes of each PSECT; PSECT names and attributes are listed in Table 10–1.

Each module in your program is named according to the name specified in the PROGRAM, BLOCK DATA, FUNCTION, or SUBROUTINE statement used in creating the module. The defaults applied to PROGRAM and BLOCK DATA statements are *source-file-name*$MAIN and *source-file-name*$DATA, respectively.

Table 10–1: 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 variables, compiler temporary variables, 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
names	Named common blocks	PIC, OVR, REL, GBL, SHR, NOEXE, RD, WRT, LONG

You can use the CDEC$ PSECT directive to change some of the attributes of a common block. See the *VAX FORTRAN Language Reference Manual* for information on the CDEC$ PSECT compiler directive statement. Table 10–2 describes the meanings of VAX FORTRAN PSECT attributes.

Table 10–2: VAX FORTRAN 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 nonshareable
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writable or nonwritable
LONG/QUAD	Longword or quadword alignment

When the 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 *VMS Linker Utility Manual.*

10.2 Storage Allocation and Fixed-Point Data Types

The default storage unit for VAX FORTRAN is the longword (four bytes). A storage unit is the amount of memory needed to store a REAL*4, LOGICAL*4, or INTEGER*4 value. REAL*8 and COMPLEX*8 values are stored in two successive storage units; REAL*16 and COMPLEX*16 are stored in four successive 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, to receive system service output values, or to be compatible with PDP–11 FORTRAN. Either specify the /NOI4 qualifier on the FORTRAN command line or explicitly declare a variable as INTEGER*2 or LOGICAL*2. This allows you to take advantage of the VAX processor's ability to manipulate both 16- and 32-bit data efficiently.

10.2.1 Integer Data Types

VAX FORTRAN supports INTEGER*2 and INTEGER*4 data types, which occupy two and four bytes of storage, respectively. The 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 with the /NOI4 qualifier, four bytes are allocated for integer values.

10.2.1.1 Relationship of INTEGER*2 and INTEGER*4 Values

INTEGER*2 values are stored as signed binary numbers in twos complement and they occupy two bytes of storage. INTEGER*4 values are also stored as signed binary numbers in twos complement, but they 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 VAX FORTRAN usage, as shown in the following example:

```
CALL SUB(2)
```

By providing an INTEGER*4 constant as the actual argument, SUB executes correctly even if its dummy argument is typed as INTEGER*2.

10.2.1.2 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 preventing integer overflow conditions like those 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 is computed. The 16-bit addition, however, will overflow the valid INTEGER*2 range and be treated as –32766. This value is converted to INTEGER*4 type and assigned to J. The overflow will be detected and reported if the default /CHECK=OVERFLOW qualifier is specified when the program unit is compiled.

Contrast the preceding example with the following apparently equivalent program, which produces different results:

```
INTEGER*2 I
INTEGER*4 J
PARAMETER (I=32767)
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 is assigned the value 32770.

10.2.1.3 Integer-Valued Intrinsic Functions

A number of the intrinsic functions provided by VAX FORTRAN produce integer results from real arguments (for example, INT). (See the *VAX FORTRAN Language Reference Manual* for more information on intrinsic functions.) In order to support such functions in a manner compatible with both INTEGER*2 and INTEGER*4 modes, two versions of these integer-valued intrinsic functions are supplied. The compiler chooses the version that matches the /I4 qualifier setting on the FORTRAN command line, that is, /I4 or /NOI4. This process is similar to generic function selection 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 intrinsic function that is the opposite of the compiler qualifier setting. For this reason, a pair of additional intrinsic function names are provided for each standard integer-valued intrinsic 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).

10.2.2 BYTE (LOGICAL*1) Data Type

VAX FORTRAN's BYTE data type lets you take advantage of the byte-processing capabilities of the VAX processor. BYTE, or LOGICAL*1, is a signed integer data type and is useful for storing and manipulating Hollerith data.

In general, when different data 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 FORTRAN 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.

10.2.3 Zero-Extend Intrinsic Functions for Converting Data Types

VAX FORTRAN normally converts a smaller fixed-point data type to a larger fixed-point data type by sign-extending the smaller value. This means that the high-order bits of the larger data type are set to the same value as the sign bit of the smaller data type. Thus, if you are converting a BYTE value to an INTEGER*4 value, the bits of the three high-order bytes of the INTEGER*4 value are set to the same value as the sign bit of the BYTE value. Generic and specific conversion functions are provided with VAX FORTRAN:

- The generic function ZEXT allows you to zero-extend, instead of sign-extend, a value to either INTEGER*2 or INTEGER*4, depending on the setting of the /I4 qualifier in the FORTRAN command line. This means that the high-order bits of the larger data type are set to zero, rather than to the sign bit of the smaller data type.
- The specific functions IZEXT and JZEXT zero-extend a value to either INTEGER*2 or INTEGER*4, respectively. The argument to IZEXT can be any fixed-point data type that occupies one or two bytes of storage, and the argument to JZEXT can be any fixed-point data type that occupies one, two, or four bytes of storage.

You use the zero-extend functions primarily for bit-oriented operations. The following is an example of the use of the ZEXT function:

```
INTEGER*2 W_VAR /'FFFF'X/
INTEGER*4 L_VAR
L_VAR = ZEXT(W_VAR)
```

This example stores an INTEGER*2 quantity in the low-order 16 bits of an INTEGER*4 quantity, with the resulting value of L_VAR being '0000FFFF'X. If the ZEXT function had not been used, the resulting value of this example would have been 'FFFFFFFF'X because W_VAR would have been converted to the left-hand operand's data type by sign extension.

When you are using the zero-extend intrinsic functions, it is important to remember that integer constants in the range of -32768 to 32767 are INTEGER*2. Therefore, JZEXT(-1) is equal to 65535. The storage requirements for integer constants are never less than two bytes. Integer constants within the range of constants that can be represented by a single byte still require two bytes of storage.

10.3 Iteration Count Model for Indexed DO Loops

The VAX FORTRAN DO statement has the following features:

- The control variable can be an integer or real variable.
- The initial value, step size, and final value of the control variable can be any expression that produces a result with an integer or real data type.
- The number of times the loop is executed (the iteration count) is determined at the initialization of the DO statement; it is not reevaluated during successive executions of the loop. Thus, the number of times the loop is executed is not affected by changes to the values of the parameter variables used in the DO statement.

10.3.1 Cautions Concerning Program Transportability

Some common practices associated with the use of DO statements may not have the intended effects when used with VAX FORTRAN. For example:

- Assigning a value to the control variable within the body of the loop that is greater than the final value does not always cause early termination of the loop.
- Modifying a step-size variable or a final value variable within the body of the loop does not modify the loop behavior or terminate the loop.

- Using a negative step size (for example, DO 10 I = 1,10,-1) in order to set up an arbitrarily long loop that is terminated by a conditional control transfer within the loop results in zero iterations of the loop body if /F77 is in effect. A zero step size may result in an error (refer to the iteration count computation in Section 10.3.2).

10.3.2 Iteration Count Computation

This description of how the iteration count of an indexed DO loop is calculated assumes a DO statement of the following form:

```
DO label, V=m1,m2,m3
```

(Where m1, m2, and m3 are any expressions.) Given an indexed DO loop of this form, the iteration count is computed as follows:

```
count = MAX(0,INT((m2-m1+m3)/m3))
```

This method of computation:

- Makes possible an iteration count of zero (in which case, the body of the loop is not executed).
- 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 values are used.)
- Differs from the usual FORTRAN-66 implementations: the minimum count value in that version was one and the current minimum value is zero (refer to Section A.1 for compatibility information). Thus, when the /F77 qualifier (the default) is used, the minimum count is zero, and when /NOF77 is used, the minimum count is one.

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-generated code controls the number of iterations of the loop by comparing the control variable directly with the final value.

10.4 ENTRY Statement Arguments

The association of actual and dummy arguments is described in the *VAX FORTRAN Language Reference Manual.* In general, that description suffices for most cases. However, the VAX FORTRAN implementation of argument association in ENTRY statements differs from that of some other implementations of FORTRAN.

As described in Chapter 6, VAX FORTRAN uses the reference and descriptor mechanisms to pass arguments to called procedures (for numeric and character arguments, respectively). Some other implementations of FORTRAN use the copy-in/copy-out 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 copy-in/copy-out 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 nonstandard method in copy-in/copy-out 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. 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 of its dummy arguments are defined with the values of the corresponding actual arguments, and they may be referenced on subsequent calls to the subprogram. However, it is not advisable to attempt this in programs that are to be executed on VAX FORTRAN, or on other systems that use the call-by-reference (or descriptor) method.

VAX FORTRAN 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 nonstandard references shown in the previous example, the subprogram must copy the values of the dummy arguments. For example:

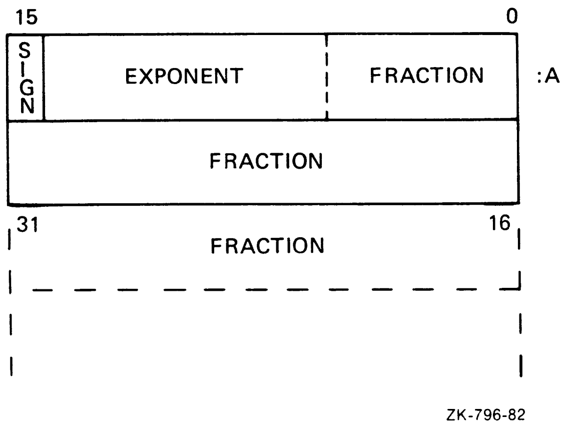
```
SUBROUTINE INIT(A1,B1,C1)
SAVE A,B,C
A = A1
B = B1
C = C1
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

Note that the use of the SAVE statement in this example ensures that the values of A, B, and C will be retained from one call to the next.

10.5 Floating-Point Data

A floating-point value is represented by 4 to 16 contiguous bytes, depending upon the specific data type. The number of exponent and fraction bits also depends upon the specific data type. The bits are numbered from right to left, 0 through n. Bit 15 is the sign bit. Figure 10–1 shows the general format of floating-point data (broken lines indicate where boundaries change, depending upon the specific data type).

Figure 10–1: General Format of Floating-Point Data



Individual floating-point data types are explained in the *VAX FORTRAN Language Reference Manual.* Data characteristics are described in Section 10.5.1.

10.5.1 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 results with VAX FORTRAN. These are described in the following sections.

10.5.1.1 Reserved Operand Faults

Accessing a floating-point variable that contains an invalid floating-point value (–0.0), indicated by an exponent field of 0 and a sign bit of 1, causes a reserved operand fault in the VAX hardware. An error is reported and, by default, your program terminates.

There are four ways to create reserved operand values:

- The VAX hardware stores a reserved operand value as the result of the floating-point arithmetic traps, 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)
  ```

  This return value can be modified with a condition handler (see Chapter 9).
- 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 causes a reserved operand fault because the integer value 32768 is a reserved operand when interpreted as a floating-point value.
- Octal and hexadecimal constants can be used to create reserved operand values.

The first two cases occur when invalid programs or data are used. The last two cases can occur inadvertently in a program and may not be detected by other implementations of FORTRAN.

10.5.1.2 Representation of 0.0

The VAX hardware defines 0.0 as any bit pattern that has an exponent field of 0 and a 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 hardware sets the fraction field to 0. One possible effect is that nonzero integers equivalenced to floating-point values may be interpreted as zero.

Logical operations can have a similar effect, as shown in the following example:

```
REAL*4 X
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 zero when interpreted as a floating-point value.

10.5.1.3 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 is not taken because the bit pattern that represents the integer value 40000 is negative (bit 15 is set) when interpreted as a floating-point value.

10.5.2 Effect of the /G_FLOATING Qualifier

The /G_FLOATING compiler qualifier causes double-precision quantities to have the G_floating type. If this qualifier is not used, the D_floating type is assumed.

Program units that exchange double-precision values should use either G_floating or D_floating data types, but not both. Because their formats are different, mixing the two types can produce unpredictable results. Under certain circumstances, however, you may want to convert one data type to the other (refer to Section 10.5.3). Passing a REAL*4 variable as an actual argument to a routine in which the corresponding dummy argument is of the G_floating data type can also give incorrect results.

10.5.3 Conversion Between D_floating and G_floating Data Types

Although you should not normally mix the D_floating and the G_floating implementations of REAL*8, there may be times when you must convert one to the other so that the values are compatible within the same program. For example, you may have a program that uses G_floating values for its double-precision work but which must access an unformatted file of D_floating values. The Run-Time Library provides procedures that perform conversions between the D_floating and the G_floating data types.

It is possible for your program to manipulate REAL*8 variables whose representations are the opposite of that specified by the /G_FLOATING qualifier. You must not perform floating-point arithmetic on such values. However, these variables can appear in unformatted I/O statements, in assignment statements, and as actual arguments.

You should be aware of the following properties of the Run-Time Library data type conversion procedures:

- Conversion from D_floating to G_floating may involve rounding of the values.
- Conversion from G_floating to D_floating is exact; there is no rounding of the values. However, if the G_floating value is not representable as a D_floating value, one of the following occurs:
  - If the value is too large, the procedure signals an overflow (and produces a floating reserved operand).
  - If the value is too small, the procedure produces a value of 0.0. If the calling procedure enabled floating underflow, the procedure signals floating underflow.
- If you attempt to convert a floating reserved operand, the procedure signals a reserved operand fault.

The Run-Time Library provides two conversion functions and two conversion subroutines.

10.5.3.1 Run-Time Library Conversion Functions

One function converts D_floating values to G_floating values and the other performs the opposite conversion. These functions have the forms:

```
MTH$CVT_D_G(D_value)

MTH$CVT_G_D(G_value)
```

D_value
Is a REAL*8 (D_floating) value to be converted to the corresponding G_floating representation.

G_value
Is a REAL*8 (G_floating) value to be converted to the corresponding D_floating representation.

You must declare these function names as REAL*8 to prevent them from being implicitly typed as integer.

10.5.3.2 Run-Time Library Conversion Subroutines

One subroutine converts arrays of D_floating values to G_floating values, and the other performs the opposite conversion. These subroutines have the forms:

```
CALL MTH$CVT_DA_GA(D_source, G_dest, count)

CALL MTH$CVT_GA_DA(G_source, D_dest, count)
```

D_source
Is an array of REAL*8 (D_floating) values to be converted to the G_floating type.

G_source
Is an array of REAL*8 (G_floating) values to be converted to the D_floating type.

G_dest,D_dest
Are the REAL*8 arrays in which the converted values are to be stored.

count
Is an INTEGER*4 value that is the number of array elements to be converted.

You can perform an in-place conversion by specifying the same array as the source and the destination argument. However, you must not specify a partial overlap between the source and destination arrays. The following calls on the conversion subroutines are valid:

```
CALL MTH$CVT_DA_GA(X,Y,100)

CALL MTH$CVT_GA_DA(X(6),X(6),95)
```

10.5.3.3 Sample Conversions

The following VAX FORTRAN program reads 100 D_floating values from an unformatted file, converts them to G_floating values, calculates a root-mean-square in G_floating, and writes the result to another unformatted file as a D_floating value.

```
OPTIONS /G_FLOATING

REAL*8 X(100), ROOT, SUM
REAL*8 MTH$CVT_G_D

READ(1) X
CALL MTH$CVT_DA_GA(X,X,100)

SUM =0.0
DO I=1,100
    SUM = SUM + X(I)**2
END DO
ROOT = SQRT(SUM/100)

WRITE (2) MTH$CVT_G_D(ROOT)
TYPE *, 'The root-mean-square is', ROOT
END
```

Chapter 11

Performance Optimization

The objective of optimization is to produce source and object programs that can achieve the greatest amount of processing with the least amount of time and memory. Realizing this objective requires programs that are carefully designed and written, and a compiler that uses compilation techniques that make optimum use of the architecture of the computer on which the programs are executed.

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. In addition, this awareness will often make it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The VAX FORTRAN compiler produces efficient code through the use of compilation techniques that take advantage of operations provided by VAX native-mode architecture and VAX hardware. The primary goal of optimization performed by the VAX FORTRAN compiler is faster execution. A secondary goal is to reduce the size of the object program.

This chapter describes optimization techniques used by the compiler and coding practices that promote optimization, both in relation to the compiler and in relation to VAX systems in general.

The chapter is divided into six major sections:

- Section 11.1 covers a variety of issues, including how optimization affects machine code and diagnostics, how the selection of source-program algorithms affect efficiency, and how the compiler functions.
- Section 11.2 discusses why debugging is best done on programs that have been compiled with the /NOOPTIMIZE qualifier on the FORTRAN command line.

- Section 11.3 describes how the compiler performs a global analysis of the variables and arrays used in a program unit. It explains how the variables are selected and the constructs and declarations in the source code that influence that selection.
- Section 11.4 describes optimizations that improve the running time of programs.
- Section 11.5 describes optimizations that decrease the memory used by programs.
- Section 11.6 provides an example that illustrates the optimizations described earlier in the chapter and demonstrates the performance enhancements.

In this chapter, much of the material in the later sections assumes that you are familiar with material in earlier sections. Understanding the material in Section 11.3 is particularly important.

11.1 General Optimization Issues

This section presents background information that you need to know in order to use your time and efforts wisely when trying to improve the run-time performance of your programs. This section provides information in the following areas:

- An explanation of how sound coding practices can have a far greater influence on efficiency than compiler optimizations.
- A description of the differences and similarities of optimized and unoptimized code.
- An overview of how the compiler operates and of the internal data structures that the compiler uses.

11.1.1 Importance of Algorithms Used in Source Programs

Solving any given problem with a computer program can usually be done in a variety of ways. The method you chose typically has more effect on the performance of the program than any amount of compiler optimization. Thus, in an effort to improve the performance of a program, it is often useful to look for a faster general method, instead of trying to make the individual operations faster.

To illustrate the importance of the algorithm used to solve a problem, consider the problem of searching for a given value in a sorted list. You can use many different methods to search for a value, and each method has advantages and disadvantages relative to the others. Each method is characterized by a certain number of fundamental operations that it must perform. For instance, the fundamental operations of a simple linear search are tests. For example:

```
C Assumes A is sorted in ascending order
C Returns the value of INDEX at which A(INDEX) .EQ. VAL
C Returns INDEX = 0 if none
C
       SUBROUTINE SEARCH(A,N,VAL,INDEX)
       INTEGER A(N),VAL
       INDEX = 0
       DO 30 I=1,N
       IF (VAL-A(I)) 10, 20, 30
10             INDEX = 0
               RETURN
20             INDEX = I
               RETURN
30     CONTINUE
       END
```

No matter how well the compiler optimizes the tests, it will take, on average, N tests (including the loop ending tests) to find the desired value. Alternatively, consider the following binary search method, which is more complicated but takes only about 2 * (LOG N (base 2)) tests:

```
      SUBROUTINE SEARCH(A,N,VAL,INDEX)
      INTEGER A(N),VAL, HIBOUND, LOBOUND
      HIBOUND = N
      LOBOUND = 1
      DO 40 WHILE (HIBOUND .GE. LOBOUND)
              I = (HIBOUND + LOBOUND)/2
              IF (A(I)-VAL) 10, 20, 30
10            LOBOUND = I + 1
              GO TO 40
20            INDEX = I
              RETURN
30            HIBOUND = I - 1
40    CONTINUE
      INDEX = 0
      END
```

Note that for N = 1000, the sequential search will average about 1000 tests in the loop, and the binary search will average about 20 (including the loop termination tests). No matter how much optimization is applied to the linear search, it will be much slower than the binary search.

Compiler optimization generally makes only the operations specified by your source program run faster; it cannot make algorithm transformations. For this reason, it is important that you try to use a faster algorithm before trying to improve the optimization of the individual operations.

11.1.2 Characteristics of Optimized and Unoptimized Programs

Optimized programs produce results and run-time diagnostic messages identical to those produced by equivalent unoptimized programs. However, an optimized program may produce fewer run-time diagnostics, and the diagnostics may occur at different source program statements or in a different order. For example:

Unoptimized Code	Optimized Code
`    A = X/Y`	`    t = X/Y`
`    B = X/Y`	`    A = t`
`    DO 10, I=1,10`	`    B = t`
`10   C(I) = C(I)*(X/Y)`	`    DO 10, I=1,10`
	`10   C(I) = C(I)*t`

The value of Y may be 0.0. Thus, the unoptimized program may produce up to 12 divide-by-zero errors at run time (if you provide code in your program to handle the error); whereas, the optimized program never produces more than one. (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 arithmetic errors; you should keep this in mind when you include error-detecting routines in your program.

11.1.3 Compiler Structure

This section describes the overall design of the VAX FORTRAN compiler and how it achieves its optimizations.

The compiler transforms your source program to VAX object code in a series of stages, called phases. Each phase transforms the program in a certain way and gathers information in preparation for the next phase. The order of the phases is chosen so that the transformations in the earlier phases increase the effectiveness of the later phases. Some phases are optional. The phases that are executed for any given program depend on the following factors: the complexity of the program, the source errors detected in the program, the use of the /NOOPTIMIZE qualifier, and the optimizations that actually take effect.

At various points in the compilation process, the program is represented in a series of intermediate forms as data structures in VAX memory. In general, these structures are considerably larger than either the original source representation or the final object representation. The actual operation of each phase consists of modifications to these internal data structures.

Note that the compiler tries to trade off slower compile time for faster run time. This limits the number and kinds of constructs that the compiler can optimize.

11.2 Effects of Compiler Optimizations on Debugging

The VAX FORTRAN compiler provides the /NOOPTIMIZE qualifier to disable many of its optimizations. This qualifier is provided to facilitate debugging of FORTRAN programs with the VMS Debugger. If the /NOOPTIMIZE qualifier is not used, many debugging commands do not work as expected.

Some of the ways that optimizations can affect debugging operations are as follows:

- *Coding order.* Some compiler optimizations cause code to be generated in an order that differs from the order in which it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when you are using the debugger to step through the code or to display the source lines.
- *Control flow.* If there are no intervening labels, the compiler assumes that statements are executed in the sequence in which they appear in source code. The compiler also eliminates label definitions if they are not needed in the machine code. As a result, you cannot reference such labels using the debugger %LABEL commands.
- *Use of condition codes.* This optimization technique takes advantage of the way in which the VAX condition codes are set. For example, consider the following source code:

  ```
  X = X + 2.5
  IF (X .LT. 0.0) GO TO 20
  ```

  To determine whether to branch, the optimized machine code uses the condition code settings after 2.5 is added to X, instead of explicitly testing the new value of X. Thus, if you attempt to set a debugging 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, deposited value of the variable does not influence whether the branch is taken.
- *Use of registers.* Some compiler optimizations make use of VAX general-purpose registers to speed up operations. When the value of a variable is being held in a register, its value in memory is generally invalid. A spurious value is often displayed if the EXAMINE command is issued for a variable under these circumstances. Sometimes the compiler is able to determine that the value of a variable is not needed in memory at all. In this case, the variable is not allocated a memory address, and double asterisks (**) are given in the address field for the variables in the memory map part of the compiler output listing. Attempting to examine such a variable will result in a warning message from the debugger.

You can avoid these problems, as well as several others, by compiling with the /NOOPTIMIZE qualifier. In general, the use of the /NOOPTIMIZE qualifier affects compile speed only marginally.

If you do not suppress optimization, you should specify the /LIST and /MACHINE_CODE qualifiers on the FORTRAN command line. This is necessary because you may need to refer to a compiler listing of the machine code generated by the compiler.

The use of some features of the VAX architecture will cause programs to violate the rules and prohibitions of the VAX FORTRAN language. In many of these cases, these rules cannot be checked efficiently at either compile time or run time, and no error messages are generated. However, the use of these features will, in some cases, interact with compiler optimizations and cause unexpected results. In most cases, you can overcome such problems by declaring the affected variables with the VOLATILE statement (see Section 11.3.2.2). While tracking down the problem, you may find it convenient to use the /NOOPTIMIZE qualifier to disable all optimizations.

11.3 Global Analysis of the Use of Variables and Arrays

The compiler optimizes an entire program unit at a time. Several of its phases are devoted to tracing the values of variables and array elements as they are created and used in different parts of the program unit. This process is called global data-flow analysis and is an important part of many optimizations. One particular usage of the language degrades the effectiveness of this process and thus prevents optimizations that would be possible if a different source construct were used to perform the same function. This usage is the assigned GOTO statement.

An assigned GOTO statement degrades optimizations because the compiler treats any labels that have been assigned to any variable as possible destinations of the GOTO. For example, consider the following program fragment:

```
        A = 5.0
        ASSIGN 10 TO LAB
         . . .
        GO TO LAB
         . . .
        ASSIGN 20 TO LAB
         . . .
20      A = X + Y
         . . .
10      B = A + 4.0
```

In this example, the compiler would try to propagate the value of 5.0 for A to the assignment to B. If this were possible, that statement could be simplified to B = 9.0. However, the second ASSIGN statement means that the compiler must assume that A can be changed before its use. If a computed GOTO statement or a simple GOTO statement were used instead of an assigned GOTO, the compiler could do the necessary analysis to allow the optimization.

11.3.1 Criteria for Selecting Variables and Arrays for Global Analysis

The global data flow analysis done by the compiler primarily consists of tracing the use of variables and arrays throughout a program unit. The speed with which this analysis can be accomplished depends partly on how many variables and arrays are to be analyzed. For this reason, the compiler puts an upper limit on the number of variables and arrays for which it performs the global analysis. The compiler uses heuristic methods to select the variables and arrays in the program unit that would benefit most from analysis. It then limits its analysis to those variables and arrays that are selected.

In general, the selection methods are biased toward the most heavily used variables and arrays. Each appearance of a variable or array in the source program counts as a use, with extra counts being given to uses inside DO loops.

11.3.2 Factors Influencing Global Analysis

The use of EQUIVALENCE statements, volatile declarations, statement functions, and variable format expressions can affect global analysis. These factors are discussed in the sections that follow.

11.3.2.1 Effects of EQUIVALENCE Statements

Quantities in EQUIVALENCE groups share memory locations. When variables and arrays are used in EQUIVALENCE statements, the optimizer must ensure that the effects of using one variable or array in the equivalence group are accounted for with regard to assignments to the other variables and arrays. For example:

```
EQUIVALENCE (A,B)
A = 4.0
 . . .
B = X + 1
 . . .
Z = A + 5.0
```

The compiler does not allow the value of 4.0 for A to be propagated to the assignment to Z because of the assignment to B.

For this reason, the variables and arrays used in EQUIVALENCE statements in each separate PSECT (common block) are grouped together and analyzed as if they were a single array during global analysis. As a result, global analysis is usually more effective if unnecessary EQUIVALENCE statements are avoided.

In particular, EQUIVALENCE statements are often used to decrease the memory requirements of programs originally written for use on computers with address spaces that are smaller than those provided by VAX systems. Because this exercise is usually unnecessary on a VAX processor, you can often improve compiler optimization in these programs by removing unnecessary EQUIVALENCE statements. In general, for best optimization, you should separate the uses of arrays and variables by the role they fill and not use the same storage location for two different roles within the program. In addition to performance benefits, the removal of the unnecessary EQUIVALENCE statements can make debugging easier.

11.3.2.2 Effects of Volatile Declarations

VAX FORTRAN provides the VOLATILE statement as an extension to the standard FORTRAN–77 language. Its purpose is to allow your programs to use certain run-time features of the VAX environment that violate the rules and prohibitions of the FORTRAN language, but at the same time take full advantage of the optimization capabilities of the VAX FORTRAN compiler.

When variables, arrays, or common blocks are declared volatile, they are never selected for global analysis. In addition, many of the local optimizations described in this section are disabled. If these optimizations are not disabled, they may cause the program to exhibit unexpected behavior. In general, this unexpected behavior will be intermittent and hard to trace. Some of the circumstances in which volatile declarations should be used are as follows:

- *When using variables in shared global common.* Using a combination of link-time and run-time options, it is possible for two different FORTRAN programs running at the same time to use shared memory to communicate. These programs may be running on a single CPU sharing standard main memory, or they may be running on different CPUs accessing a block of storage in a multiport memory unit.

  In addition, some special I/O devices can be controlled by sharing memory with a FORTRAN program. This shared memory must be declared as common blocks and must be page aligned. This usage is referred to as shared global common. This method is often the fastest way for two programs to communicate large amounts of data on VAX.

  Shared global commons should always be declared VOLATILE when the contents may change without synchronization with the program unit. See Section 8.1.2 for more information about how to use shared global commons.
- *When using variables in a condition handler.* FORTRAN programs can take advantage of the VAX condition handling mechanisms. These mechanisms can be used in various cases in which control is taken away from the normal execution order of the program and given to a special routine called a condition handler.

  When writing programs that use these exception mechanisms, you should always declare as volatile any variables or arrays that are shared between the condition handler and the program. Failure to do this may result in one of the following problems:

  - Cause the condition handler to use an incorrect value for the shared quantity
  - Cause the program to ignore handler modifications of the shared quantity

  See Chapter 9 for more information about the use of the VAX exception handling mechanisms.

- *When remembering the addresses of arguments in a subroutine.* Some subprograms have the capability to retain the addresses of their arguments after they have returned, and can reuse those addresses in later calls even though the arguments do not explicitly appear in the argument list for those calls. When calling such subprograms, you should always declare the arguments whose addresses are retained as volatile. This prevents the compiler from holding them in registers across later calls.

  Most often, this situation applies to those systems of routines for which you must make an initialization call with an array to be used for "working storage" while the system is being used. The arrays used for working storage should be declared as volatile.

11.3.2.3 Effects of Inline Expansion of Statement Functions

The compiler attempts to expand statement functions directly inline, instead of calling them as functions. This has several performance benefits:

- It allows the expression inside the statement function to participate in the global analysis performed on the rest of the program.
- It saves the overhead of a subprogram call and return.
- It eliminates both the need to store the actual arguments in memory before the call and the need to retrieve them from memory after the call.

For example:

Unoptimized Code	Optimized Code
`SUM(A,B) = A + B`	`SUM(A,B) = A + B`
...	...
`Y = 3.14`	`Y = 3.14`
`X = SUM(Y,3.0)`	`X = 6.14`

You can determine from a compiler output listing whether a statement function is always expanded inline. For program units that contain statement functions, a summary of the statement functions is printed at the bottom of the listing. The address and data type of each statement function is listed. If double asterisks (**) appear in the address field, the statement function was expanded inline each time it was referenced.

Not all statement functions can be expanded inline, however. The following conditions limit the compiler's ability to expand statement functions inline:

- *Use of excessively large code segments.* If the body of the statement function generates a large-sized code sequence, it will not be expanded. This is because large statement functions benefit relatively less from the advantages of expansion discussed previously. Also, expanding large statement functions can cause the calling program to grow to an excessive size. The cutoff point is approximately twenty machine instructions; it varies depending on data type and operation complexity.
- *Use of external function calls.* If the body of the statement function contains a call to an external function or to another unexpanded statement function, it will not be expanded. Again, this is because such functions benefit relatively less from the expansion than do other functions.
- *Use of CHARACTER operations.* If the function returns a CHARACTER result or uses any CHARACTER variables, arrays, operations, or arguments, it will not be expanded. This is because character operations on VAX processors destroy the contents of several registers and would significantly degrade other optimizations if expanded inline.
- *Incorrect use of arguments.* If the actual arguments do not match the formal arguments to the statement function in order, number, or data type, it will not be expanded. Some usages generate a warning message, such as a mismatch in the number of arguments. Others generate correct results if called, but not if expanded. In these cases, no expansion is done, but a warning is not given.

If a statement function is not expanded inline, it degrades other optimizations besides calls to itself. In particular, it prevents any local variables used in the statement function body from being selected for global analysis. For this reason, you should limit the use of local variables in statement functions to those cases that match the criteria described in the preceding list.

11.3.2.4 Effects of Variable Format Expressions

VAX FORTRAN provides variable format expressions (VFEs) for increased flexibility and performance of formatted I/O (see Section 11.4.5). Using VFEs is generally preferable to using run-time format expressions because it allows the compiler to use the efficient compile-time format interface to the FORTRAN RTL I/O support routines. Using a variable in a VFE, however, does prevent the variable from being selected for global analysis. For this reason, variables used in VFEs should not be used elsewhere.

11.4 Speed Optimizations

Speed optimizations reduce the running time of programs. They fall into four categories:

- *Removal optimizations.* Remove unnecessary operations from the program.
- *Replacement optimizations.* Make necessary operations more efficient.
- *Operation-specific optimizations.* Apply to certain individual operations.
- *I/O optimizations.* Reduce I/O system overhead and are controlled by how you set up I/O operations in your source program.

Sections 11.4.2 through 11.4.4 provide detailed information about these optimizations.

Being aware of these optimizations may make it easier for you to write your programs in a straightforward and general way. In many cases, you can also gain a performance benefit by modifying an existing source program to take advantage of them, as shown by the examples, but this practice can often lead to errors and makes the program harder to read.

11.4.1 Effects of Global Analysis on Speed Optimizations

Global analysis is required for both removal and replacement optimizations.

For removal optimizations, both arrays and variables must be analyzed. The upper limit on the number of variables and arrays that the compiler will analyze for these optimizations is 128. If the number of variables and arrays in the program unit is more than 128, the compiler selects 126 of the variables and arrays that are most heavily used, and treats all remaining local variables and arrays as if they were a single array and all remaining COMMON variables and arrays as if they were a different, single array.

For replacement optimizations, only variables need to be analyzed, not arrays. However, because replacement optimizations require additional analysis beyond that required for removal optimizations, the compiler separately selects a set of variables for this additional analysis. The maximum number of variables for this kind of analysis is 32. (Most replacement optimizations involve the use of the VAX general-purpose registers. Because the number of registers available for use by optimizations is 13, selecting additional variables beyond 32 would not generally result in more replacement optimizations.)

The selection criteria for variables used for this analysis is more restrictive than those for removal optimizations. In particular, variables whose usage prevents them from being loaded into registers are not selected. These include variables used in common blocks, variables used in EQUIVALENCE statements, variables declared volatile, variables used in statement functions that do not get expanded in-line, and variables used in variable format expressions (see Section 11.3.2).

11.4.2 Removal Optimizations

Because removal optimizations eliminate run-time operations from programs, their effect on program performance is generally greater than that of replacement optimizations. The following sections describe some of the removal optimizations.

11.4.2.1 Compile-Time Operations

The compiler attempts to perform as many operations as possible at compile time. This removes them entirely from the object program.

Constant Operations

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, the following statement is computed as a single move instruction:

```
X = -10.0
```

- *Arithmetic operators on integer, real, and complex constants.* Expressions involving +, −, *, or / operators are evaluated at compile time. For example:

Unoptimized Code	Optimized Code
`PARAMETER (NN=27)`	`PARAMETER (NN=27)`
`I = 2*NN + J`	`I = 54 + J`

  Evaluation of some constant functions and operators is performed at compile time. In particular, the CHAR, ABS, MAX, MIN, and MOD functions of constants, concatenation of string constants, and logical and relational operations involving constants are performed at compile time.

- *Type conversions of constants.* Lower-ranked constants are converted to the data type of the higher-ranked operand at compile time. For example:

Unoptimized Code	Optimized Code
`X = 10 * Y`	`X = 10.0 * Y`

- *Array address calculations.* Array address calculations involving constant subscripts are simplified at compile time whenever possible. For example, the following statement is compiled as a single move instruction:

```
DIMENSION I(10,10)
I(1,2) = I(4,5)
```

Initialization of Argument Lists

Argument lists are initialized at compile time whenever the address of the argument can be determined at compile time. For example, the following statement is compiled as a single CALL instruction.

```
CALL SUBX(M, A(3,4))
```

(See Section 6.1.3 for additional information about argument lists.)

Delaying Optimizations

The compiler eliminates operations that can be shown at compile time to have no effect, or that can be transformed so as to be unnecessary.

For example, a number of operations can be mathematically proven to leave the result unchanged. These operations are eliminated. For example, the following statement is compiled as a simple move instruction:

```
X = Y * 1.0
```

It is also possible to eliminate more complicated combinations. For example, the following statement is compiled as a move negated instruction:

```
X = Y * -1.0
```

Another example of operations that can be shown to have no effect at compile time involves some unary minus and .NOT. operations. These operations can also be delayed until they can be proven to be unnecessary, and if so, they are eliminated. In the following example, both negations are eliminated:

```
X = -Y * -Z
```

11.4.2.2 Flow Boolean Operations

In general, it is not necessary to actually generate a temporary logical variable with the value of the logical expression used in an IF statement. In most cases, the compiler avoids generating these unnecessary temporary variables. Instead, it makes use of compare operations and condition codes that are a more efficient means of controlling program behavior.

11.4.2.3 Compound Logical Expressions in IF Statements

Unnecessary operations in compound logical expressions in IF statements can often be avoided. Many compound logical expressions do not need to be evaluated completely and a partial evaluation suffices to determine the final outcome. For example, if e1 in the following expression has the value .FALSE., e2 is not evaluated:

```
IF (e1 .AND. e2) GO TO 20
```

Effectiveness

In addition, the compiler changes the order of evaluation of the components of the expression to do the simplest first. Thus, in the following example, the subexpression X .GT. Y is evaluated first, and if it evaluates .TRUE., the array element comparison is not performed:

```
IF (A(I,J) .GT. B(M,N) .OR. X .GT. Y) GO TO 20
```

Correctness

The FORTRAN-77 language prohibits programs that depend on the order of evaluation of their subexpressions. This dependency can arise from the occurrence of side effects within compound logical expressions in IF statements, as described previously. For example, if the following statement was used repetitively, the function RAN would be called during some executions but not on others because the compiler would evaluate A .GT. B first and would avoid the RAN call if it evaluates as .FALSE.:

```
IF (RAN(K) .GT. 0.5  .AND. A .GT. B) GO TO 20
```

Because the RAN call has a side effect on its argument, different sequences of random numbers will be generated depending on the sequence of values of A and B. This affects the control flow of the program and, as a result, different answers may be generated (that is, answers different from those that would be generated if RAN(K) were always evaluated).

For this reason, you should always explicitly reference functions with side effects prior to their use in logical expressions. For example, the preceding program could be made to conform to the language rules as follows:

```
RANVAL = RAN(K)
IF (RANVAL .GT. 0.5 .AND. X .GT. Y) GO TO 20
```

11.4.2.4 Common Subexpression Elimination

The same subexpression often appears in more than one computation within a program unit. 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 preceding sequence is compiled as follows:

```
t = B*C
A = t + E*F
 . . .
H = A + G + t
 . . .
IF ((t)-H) 10,20,30
```

Note that two computations of B*C have been removed. This optimization is called common subexpression elimination.

Of course, you could have optimized the source program yourself to avoid the redundant calculation of B*C. The following example shows a more significant application of this kind of compiler optimization. In this case, you could not reasonably modify the source code to achieve the same effect. Consider the following statements:

```
DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)
```

Without optimization, these statements can be compiled as follows:

```
t1 = (J-1)*25+I
t2 = (J-1)*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-1)*25+I
A(t) = B(t)
```

11.4.2.5 Code Motions

Execution speed is enhanced by taking invariant computations out of loops. This optimization is called loop hoisting. Loop hoisting can be unconditional or conditional.

Unconditional Loop Hoisting

If the compiler detected the following sequence, it would recognize that the subexpression 3.0*Q has the same value each time the loop is executed:

```
      DO 10, I=1,100
10    F = 3.0*Q*A(I) + F
```

Thus, it would change the sequence as follows:

```
      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, thus saving 99 multiply operations.

Conditional Loop Hoisting

Moving code out of loops is possible even if some of the code is not always executed in each iteration of the loop. For example:

```
      DIMENSION A(25,25), B(25,25)
       . . .
      DO 10 I=1,25
      IF (I .GT. K) THEN
              A(I,J) = A(K,J)
      ELSE
              B(I,J) = B(K,J)
      ENDIF
10    CONTINUE
```

In this case, the subscript computations can be moved out of the loop, and the preceding sequence is compiled as follows:

```
      t1 = (J-1)*25
      t2 = t1+K
      DO 10 I=1,25
      IF (I .GT. K) THEN
              A(t1+I) = A(t2)
      ELSE
              B(t1+I) = B(t2)
      ENDIF
10    CONTINUE
```

This example shows that all of the multiplications and half of the additions used in array addressing computations are eliminated from the loop by conditional hoisting.

11.4.2.6 Value Propagations

The compiler keeps track of the values assigned to variables and traces the values to all of the places that they are used. If it is more efficient to use the value than the variable, the compiler makes this change. This optimization, called value propagation, can have several beneficial effects.

The following sections describe the propagation of variables, the propagation of constants, and the elimination of variables.

Propagation of Variables

One beneficial effect is to remove unnecessary memory references from the program. For example:

Unoptimized Code	Optimized Code
`A = Z`	
`...`	`...`
`X = F(X) + A`	`X = F(X) + Z`

Note that two operations have been removed from the program: storing Z into variable A and retrieving A from memory.

Propagation of Constants

Additional improvements are possible when the quantity being propagated is a constant known at compile time. This special case is called constant propagation.

Constant propagation has several benefits:

- Run-time operations can be replaced with compile-time operations. For example:

Unoptimized Code	Optimized Code
`PI = 3.14`	`PI = 3.14`
`...`	`...`
`PIOVER2 = PI/2`	`PIOVER2 = 1.57`

In this case, the divide operation has been removed from the program. This process is repeated and further constant propagations are often possible.

- Comparisons and branches can be avoided at run time. For example:

Unoptimized Code	Optimized Code
`I = 100`	`I = 100`
`...`	`...`
`IF (I .LT. 1) GO TO 100`	`A(100) = 3.0*Q`
`A(I) = 3.0*Q`	

- In some cases, several statements can be eliminated. For example:

Unoptimized Code	Optimized Code
`        I = 100`	`        I = 100`
`         . . .`	`         . . .`
`        IF (I .GT. 1) GO TO 10`	`10      Y = 3.0*Q`
`        M = N*J`	
`        K = M + I`	
`10      Y = 3.0*Q`	

The statements immediately following the test are removed because they can never be executed.

In addition to propagating constant values from program assignments, the compiler propagates constant values from DATA statements. When compiling subprograms, the compiler analyzes the program to ensure that this is done correctly if the subroutine is called more than once. For example:

Unoptimized Code	Optimized Code
`SUBROUTINE SUBA(K,M,N)`	`SUBROUTINE SUBA(K,M,N)`
`DATA I,J/3,4/`	`DATA I,J/3,4/`
`M = I * 4`	`M = 12`
`N = J * 4`	`N = J * 4`
`IF (K .GT. 0) J = K`	`IF (K .GT. 0) J = K`
`...`	`...`

The value of 3 is propagated from the DATA initialization of I, eliminating a multiply operation. The value of 4 for J is not propagated because it may not retain its initialization after the first call to SUBA.

Variable Elimination

Occasionally, value propagation can eliminate the need for a variable. You can determine whether a variable has been eliminated from the program by looking at the storage map section of the compiler output listing. If the memory address of the variable is given as double asterisks (**), the variable has been eliminated.

11.4.2.7 Dead Store Elimination

Sometimes a variable is assigned a value, but that value is never used in the program. In this case, the assignment is eliminated as well as any calculation that the assignment requires. For example:

Unoptimized Code	Optimized Code
`X = Y*Z`	`...`
`...`	`X = A(I,J) * PI`
`X = A(I,J) * PI`	

If the first assignment to X is never used in the program, it is removed. If this causes previous assignments to Y and Z in the program to be unnecessary, they are removed also. This process is continued until all unnecessary operations are removed.

NOTE

> Some programs used for performance analysis contain such unnecessary operations. When attempting to measure the performance of programs compiled with VAX FORTRAN, you should not use such programs because they will give unrealistic results. Programs used for performance evaluation should always use their results in output statements or subprogram calls.

11.4.3 Replacement Optimizations

Many well-written programs contain few opportunities for removal optimizations, as described in Section 11.4.2. So, although these optimizations have a large effect each time they are used, they may provide little benefit to your program. On the other hand, replacement optimizations usually make only a modest improvement in program efficiency each time they are used, but are much more frequently usable. For this reason, replacement optimizations can produce more overall improvement than removal optimizations.

In general, replacement optimizations do not lend themselves well to being illustrated in source examples, unlike the removal optimizations discussed previously. Often, replacement optimizations can be described only in technical terms specific to the VAX architecture. For this reason, you should be familiar with the *VAX Architecture Handbook* in order to fully understand the material in this section.

NOTE

> In some examples, the code listed under the heading "optimized code" is not the exact code that would be generated by the compiler. Actual machine code may vary depending on optimizations elsewhere in the program. The examples are only intended to help explain the optimizations being discussed.

The following sections describe the various areas in which replacement optimizations take effect.

11.4.3.1 Store Delaying Optimizations

The FORTRAN language specifies that expressions in assignment statements are to be completely evaluated before the assignment is performed. In most cases, however, the same answer results on VAX processors if the appropriate 3 operand form of the operation is performed. In these cases, the compiler will use the 3 operand form of the operation to eliminate the need for a temporary variable, and a separate move instruction to put the result into its proper place. For example, consider the following source code:

```
X = 3.0*Q
```

This statement results in the following object code:

Unoptimized Code	Optimized Code
`MULF3    #3.0, Q, R0`	`MULF3    #3.0, Q, X`
`MOVL     R0, X`	

11.4.3.2 Register Usage

For most programs, the compiler is able to generate code that uses the VAX general-purpose registers instead of ordinary memory locations. Because operations using the registers are often much faster than equivalent operations that reference memory locations, such programs are much faster than equivalent unoptimized programs. The remainder of this section describes the optimizations that make use of the VAX general-purpose registers.

Using Registers to Hold Temporary Operation Results

The VAX general-purpose registers are most frequently used to hold the values of temporary results of subexpressions, even if the /NOOPTIMIZE qualifier is used. Registers can be used in many other ways to speed up programs, but no more than 13 of them are available at any point during the program; as a result, it is important to minimize using them to hold expression results. The compiler does this by reordering the evaluation of complicated expressions so as to reuse the same registers as much as possible during the evaluation. The compiler always computes the most complicated subexpression first in an operation and then, for the next operation, reuses the registers that are no longer needed. For example, consider the following statement:

```
A = B*C + D*E + F*G
```

This statement results in object code that uses only two registers:

```
MULF3   C, B, R0
MULF3   E, D, R1
ADDF2   R1, R0
MULF3   G, F, R1
ADDF3   R1, R0, A
```

Using Registers to Hold Variables

The compiler uses registers to hold the values of program variables whenever the FORTRAN language does not require them to be in memory. The compiler may hold the same variable in different registers at different points in the program. For example:

```
V = 3.0*Q
 . . .
X = SIN(Y)*V
 . . .
V = PI*X
 . . .
Y = COS(Y)*V
```

The compiler may choose one register to hold the first use of V and a different register to hold the second. Both registers may be used for other purposes at points in between. Thus, there may be points in time when the value of the variable does not exist anywhere. If the value of V is never needed in memory, it will be eliminated from the program entirely. As with variables eliminated by value propagation, double asterisks (**) are given for these variables in the memory map part of the listing.

Using Registers to Index into Arrays

Often variables are held in registers to index into arrays. For example, consider the following statement:

```
A(I) = B(J) + C(K)
```

This statement results in the following code (in the absence of other optimization effects, such as loops involving I, J, or K):

```
MOVL    I, R12
MOVL    K, R0
MOVL    J, R1
ADDF3   C-4[R0], B-4[R1], A-4[R12]
```

Most expressions, however, do not use all different index variables. For example, the following statement is more representative of normal usage:

```
A(K) = B(K) + C(K)
```

In this case, K will be loaded into only one register and will be used to index into all three arrays at the same time. The compiler will do this even if K must normally be held in memory (if K is in shared global common, for instance). The optimized code is as follows:

```
MOVL    K, RO
ADDF3   K-4[RO], B-4[RO], A-4[RO]
```

Using Base Registers for Arrays and Common Blocks

In most cases, shrinking the size of the code generated will also increase the speed with which the code is executed. Thus, it is important not only to minimize the number of operations performed, but to use the minimum size for the operand specifications involved in the operations. The FORTRAN compiler uses the general-purpose registers, called base registers, for this purpose.

Reducing the size of operand specifications can often be accomplished by loading a register with either the memory address of the operand or an address close to the operand. Then, the operand specifier can use a small offset, 0, 1, or 2 bytes, rather than a large 4-byte offset that is used when base registers are not used. Base registers are used for addressing local variables and arrays, common blocks, and dummy arrays. For example, consider the following source code:

```
COMMON  /C1/ A(10), B(10), C(10)
 . . .
A(I) = B(I) + C(I)
```

These source statements result in the following object code:

Unoptimized Code

```
 . .
MOVL    I, RO
ADDF3   C-4[RO], B-4[RO], A-4[RO]
```

Optimized Code

```
MOVAL   A, R12
 . . .
MOVL    I, RO
ADDF3   C-4(R12)[RO], B-4(R12)[RO], A-4(R12)[RO]
```

In the unoptimized case, the ADDF3 takes 19 bytes. In the optimized case, it takes seven bytes.

11.4.3.3 Using Autoincrement and Autodecrement Mode Addressing

In order to reduce both memory requirements and execution time, the compiler also uses base registers in an additional way. Specifically, the compiler can make use of the autoincrement and autodecrement register modes available to operand specifiers in the VAX architecture. The compiler uses these modes when it detects that a pointer to an array can be updated at the same time that the array is referenced. For example, consider a simple summing loop such as the following:

```
        DO 10 K=1,1000
10      X = X + A(K)
```

These statements result in the following object code:

Unoptimized Code			Optimized Code		
	MOVL	#1, K		MOVL	#1, K
L$1:	MOVL	K, RO		MOVAL	A, RO
	ADDF2	A-4[RO], X	L$1:	ADDF2	(RO)+, X
	AOBLEQ	#1000, K, L$1		AOBLEQ	#1000, K, L$1

In general, the array reference will need to be recomputed each time around the loop. This process can be greatly speeded up if, each time the array is referenced, the base register pointing to the array can be updated so that it always points to the correct element. The autoincrement and autodecrement address modes can accomplish this.

The following requirements must be met before the autoincrement or autodecrement modes can be used:

- The array index expressions cannot be too complicated; they must be simple additions or subtractions of the loop index variable with expressions that do not change within the loop.
- The successive array references within the loop must occur according to the order of subscript progression, as defined by the description of arrays within the *VAX FORTRAN Language Reference Manual*. In general, this means that the leftmost subscript should vary the fastest (that is, be in the innermost loop).
- The increment value of the loop must be a constant (a PARAMETER constant can be used).

- The compiler can use autoincrement even if the loop steps by more than one increment at a time. However, there must be at least as many array references within the loop as the increment value. For instance, autoincrement cannot be used for the following loop:

  ```
          DO 10 I=2,100,2
  10      X = X + B(I)
  ```

  Autoincrement cannot be used because the loop contains only one reference to B; this allows only one opportunity to update the base register, and autoincrement requires two opportunities. This means that you can freely use "loop unrolling" techniques, for reducing loop overhead, without sacrificing efficiency of array index calculation.

The following example shows a loop (unrolled) that results in a substantial speedup because of the use of autoincrement address modes for the array references:

Source Code:

```
      DO 50 I = MP1,N,4
        DY(I) = DY(I) + DA*DX(I)
        DY(I + 1) = DY(I + 1) + DA*DX(I + 1)
        DY(I + 2) = DY(I + 2) + DA*DX(I + 2)
        DY(I + 3) = DY(I + 3) + DA*DX(I + 3)
50    CONTINUE
```

Optimized Code:

```
      MOVL    MP1, I
      MOVAF   DX(I), R3
      MOVAF   DY(I), R4
L$1:  MULF3   (R3)+, DA, R5
      ADDF2   R5, (R4)+
      MULF3   (R3)+, DA, R5
      ADDF2   R5, (R4)+
      MULF3   (R3)+, DA, R5
      ADDF2   R5, (R4)+
      MULF3   (R3)+, DA, R5
      ADDF2   R5, (R4)+
      ACBL    N, #4, I, L$1
```

In this example, each of the array references in the loop uses an autoincrement mode. Note that the successive references to the arrays use the same or increasing array indexes. This is recommended because it allows the most autoincrement references and the smallest offset values. The compiler is, however, equally effective at optimizing loops in which the array indexes are successively decreasing rather than increasing.

In the case of decreasing array indexes, autodecrement address modes are used and you should use monotonically decreasing array indexes for maximum effectiveness. It is even possible to use both modes in the same loop, but this requires the compiler to use multiple registers for indexing because the same register cannot be used simultaneously for both autoincrement and autodecrement.

11.4.3.4 Strength Reduction Optimizations

If autoincrement or autodecrement address modes cannot be used for array addressing in a loop, it is still possible to speed up part of the addressing computation. Because the technique for doing this involves introducing an add operation in place of a multiply, it is called "reduction in operator strength," or simply "strength reduction." Basically, the technique involves using an add instruction to update an array pointer each time around the loop instead of recomputing the array index when it requires a multiply. For example:

```
        REAL*8 A(25,100)
 . . .
        V = 0
        DO 10 I=1,100
10      V = V + A(K,I)
```

These statements result in the following object code:

Unoptimized Code			Optimized Code		
	CLRD	V		CLRD	V
	MOVL	#1, R12		MOVL	#1, R3
L$1:	MULL	#25, R12, R0		MOVL	K, R12
	ADDL2	K, R0		MOVAD	A-8[R12], R4
	ADDD2	A-208[R0], V	L$1:	ADDD2	(R4), V
	AOBLEQ	#100, R12, L$1		ADDL2	#200, R4
				AOBLEQ	#100, R3, L$1

In the preceding example, the unoptimized code sequence computes successive values of A(K,I) by using MULL3 and ADDL2 operations and a context index register mode for the ADDD2. In the optimized code sequence, the MULL3 and ADDL2 are replaced by a single ADDL2 that updates the pointer into A. It also allows the reference in the ADDD2 instruction to use the shorter, indirect mode instead of context indexing.

11.4.3.5 Tradeoff Policy Applied to Register Use

In a large program, there are usually more quantities that would benefit from being in registers than there are registers to hold them. In this case, the compiler must apply tradeoffs in determining how to use the registers. In general, the compiler tries to use the registers for temporary operation results first, including array indexes; then for variables; then for base registers; then for all other usages. The decision criteria are complicated, however, and this hierarchy may not always be followed.

In general, finding the best choice of register usages for a given program requires a very large amount of computation. Essentially, the only way to do this is to try all of the methods and choose the best. If this approach was used, however, it would lead to unacceptably long compile times. For this reason, the compiler uses heuristic algorithms, and a modest amount of computation, to attempt to determine a good usage for the registers. Thus, the compiler will often not choose the best way to apply the registers, but it will normally use them in a way that achieves results significantly faster than that of an unoptimized program.

11.4.3.6 Block Moves and Block Initializations

Occasionally, statements in loops are well adapted to the use of special VAX instructions for moving a large amount of data in one operation. Typically, the data movement operations involve assigning one array to another (block moves) or filling an array with a uniform initial value (block initializations). Block moves and block initializations take advantage of the VAX MOVC3 and MOVC5 instructions, respectively. When the compiler detects these usages, it generates the appropriate special VAX instruction. For example, consider the following loop:

```
        DO 10 I=1,1000
10      A(I) = B(I)
```

These statements result in the following object code:

Unoptimized Code	Optimized Code
`      MOVL    #1, I`	`MOVC3   #4000, B, A`
`L$1:  MOVL    I, R0`	`MOVL    #1001, I`
`      MOVL    B-4[R0], A-4[R0]`	
`      AOBLEQ  #1000, I, L$1`	

Another typical usage is initializing an array with zeros. For example:

```
       DO 10 I=1,1000
10     A(I) = 0.0
```

These statements result in the following object code:

Unoptimized Code	Optimized Code
`      MOVL    #1, I`	`MOVC5  #0, (SP), #0, #4000, A`
`L$1:  MOVL    I, R0`	`MOVL   #1001, I`
`      MOVL    #0.0, A-4[R0]`	
`      AOBLEQ  #1000, I, L$1`	

A program must meet the following requirements before the compiler can use these special instructions.

- The array addresses used in the loop must be uniformly increasing by one element on each iteration of the loop.
- Other than their assignment or initialization, the arrays involved must not appear in the loop or have any dependences in the loop.
- The arrays must not overlap in memory. The most common overlap cases are the assignment of one dummy array to another, or to or from an array in COMMON. These overlap cases may occur in some calls but not others. Because the potential exists, the optimization cannot be performed in these cases.

As long as these requirements are met, even subsections of arrays can be assigned using MOVC3. For example:

```
       DOUBLE PRECISION A(1000,1000), B(1000,1000)
       INTEGER M(1000), N(1000)
        . . .
       DO 10 I=201,800
       A(I-200,J) = B(I+200,K)
10     M(I-200) = N(I+200)
```

These statements result in the use of a single MOVC3 instruction to assign 600 elements of B starting at B(401,K) and ending at B(1000,K) to elements of A starting at A(1,J) and ending at A(600,J). They also result in the use of another MOVC3 instruction to assign N(401) through N(1000) to M(1) through M(600). Any number of such assignments in the same loop can be replaced by MOVC3 instructions as long as the assignments do not have dependencies on each other.

Use of the MOVC5 instruction has some additional limitations. It can only initialize arrays in the following ways:

- Floating-point and complex arrays can only be initialized to 0.0 and (0.0,0.0), respectively.
- Integer and logical arrays can be initialized with 0 or −1.

11.4.3.7 Locality of Reference

The virtual memory architecture of the VAX series of computers allows the use of very large arrays without the need for overlays or for large amounts of physical memory equivalent to the size of the program. This is accomplished by storing the pages of memory on disk when they are not being used, and then copying them into memory when their values are needed or they are being assigned. The machines are designed to make this copying activity totally transparent and very fast. Usually, it is so fast that it is not a significant contributor to the running time of a VAX FORTRAN program.

Most programs do not reference memory addresses randomly; instead, they usually reference groups of addresses that are close to each other. This typical clustering of address references is called locality of reference.

One of the ways that the VAX series of computers makes the required copying efficient is by taking advantage of locality of reference. Whenever a given page of memory requires copying, the system also copies the pages near it, which reduces the need for future copies. This technique is called page clustering. You can improve the speed of your programs by taking advantage of page clustering. This can be done by avoiding references to widely scattered memory addresses.

Order of Subscript Progression with Array References

In FORTRAN programming, certain coding practices defeat the page clustering technique and greatly increase the amount of copying to and from disk. The most common of these inefficient practices is the referencing of arrays in an order that does not correspond to the order of subscript progression. The following example demonstrates good coding practice:

```
        DIMENSION A(1000,1000), B(1000,1000)
         . . .
        DO 10 J=1,1000
        DO 10 I=1,1000
10      A(I,J) = B(I,J)*X
```

The resulting object code executes very efficiently on VAX processors because each reference to A and B is adjacent in memory to the previous one, thus allowing page clustering to drastically reduce the number of disk copy operations.

The following example of superficially similar loops demonstrates poor coding practice:

```
        DO 10 I=1,1000
        DO 10 J=1,1000
10      A(I,J) = B(I,J)*X
```

These loops execute much less efficiently on VAX processors than the previous loops because each reference to A and B is about 1000 elements away from the previous reference. In this case, the running time of the program will be dominated by the copying of the arrays to and from the system disk.

To improve the performance of a program that suffers from excessive page faults, it is important to understand the concepts of locality of reference, page clustering, and the order of subscript progression of arrays. Then, when you identify a program with excessive page copying, you will be able to modify the program in a way that eliminates the problem.

11.4.4 Operation-Specific Optimizations

A number of speed optimizations do not fall into the categories described previously. These are described briefly in Sections 11.4.4.1 through 11.4.4.6. In general, they are ways of implementing specific VAX FORTRAN usages to take better advantages of the VAX architecture.

11.4.4.1 Constants as Code Literals

Constants used in operations are most often inserted into the executable object code directly as literal operand specifiers. For example, consider the following source code:

```
I = 14
```

This statement results in the following object code:

Unoptimized Code	Optimized Code
`K$1:  .LONG   14`	`MOVL    #14, I`
`      MOVL    K$1, I`	

11.4.4.2 JSB for Floating Math Functions

Optimized calling sequences are used for the REAL*4, REAL*8, and REAL*16 versions of some intrinsic functions. For example, consider the following source code:

```
X = SIN(Y)
```

This statement results in the following object code:

Unoptimized Code	Optimized Code
`CALLG   arglist, MTH$SIN`	`MOVF    Y, R0`
	`JSB     MTH$SIN_R4`

11.4.4.3 Code Alignment

Labels used as the objects of frequent branch instructions are aligned on longword boundaries to improve the speed of the branch operations. This optimization results in NOP instructions in the generated code. The NOP instructions generally appear at loop tops and before labels generated by the compiler to implement ELSE or ELSE IF constructs.

11.4.4.4 SIN and COS Functions

When the SIN and COS functions (or SIND and COSD functions) are both referenced using the same argument, the compiler uses an optimized calling sequence, which computes both using a single call.

11.4.4.5 Mixed Real/Complex

Operations on COMPLEX data types are optimized if the other operand is REAL. Normally, such an operation is performed by converting the REAL to a COMPLEX and then performing the operation using the two COMPLEX quantities. The optimization avoids the conversion and performs a simplified operation. The compiler performs this optimization on the +, –, and * operations if either operand is REAL, and on the / operation if the right operand is REAL. For example, consider the following source code:

```
COMPLEX A, B
 . . .
B = A + R
```

These statements result in the following object code:

Unoptimized Code	Optimized Code
`MOVF    R, R0`	`ADDF3   R, A, B`
`MOVF    #0, R1`	`MOVF    A+4, B+4`
`ADDF3   R0, A, B`	
`ADDF3   R1, A+4, B+4`	

11.4.4.6 Peephole Optimizations

The final representation of the intermediate code is examined on an instruction-by-instruction basis to find operations that can be replaced by shorter, faster operations. For example, consider the following source code:

```
A = 0.0
```

This statement results in the following object code:

Unoptimized Code	Optimized Code
`MOVF    #40, A`	`CLRF    A`

The two most common peephole optimizations are test elimination and conversion of operations from a 3 operand form to a 2 operand form.

These optimizations are described in the remainder of this section.

Test Elimination

IF statements can frequently be optimized when the variables used in their test conditions are computed immediately before the IF statement. For example, consider the following source code:

```
 . . .
I = M*N
IF (I .GT. 0) THEN
 . . .
```

In this case, because the value of I is already indicated by the VAX condition codes at the time the IF statement is executed, it does not need to be explicitly compared with zero. When the compiler detects this situation, it eliminates the compare operation.

Conversion of 3 Operand Form to 2 Operand Form

Many VAX arithmetic and logical operations exist in both 2 operand and 3 operand forms. The 3 operand forms are generally used because they often prevent the need for MOV instructions (see Section 11.4.3.1). However, in many cases in which both input and output operands are the same register or memory location, the 2 operand form can be used instead. The methods used to choose locations for the results of temporary operations are designed to take advantage of this possibility. The 2 operand form has advantages in both execution time and memory space. For example, consider the following source code:

```
A = A + B
```

This statement results in the following object code:

Unoptimized Code	Optimized Code
`ADDF3   B, A, A`	`ADDF2   B, A`

11.4.5 Improving Performance of I/O Operations

Many FORTRAN programs spend more time and resources performing input and output operations than they do performing computations. In these programs, making the I/O operations more efficient is more worthwhile than making the computations more efficient. This section discusses some techniques that you can use to make your I/O operations more efficient.

11.4.5.1 Using Unformatted I/O

FORTRAN formatted input and output operations are often compute bound. That is, they often spend as much time converting input characters into internal form and the internal form into output characters as they do performing the actual data transfers to and from the output device. For this reason, you should limit your use of formatted I/O to those situations in which it is necessary for someone to provide input or to examine the output.

In cases where the input comes from another program or where the output is processed by another program, you should use unformatted I/O instead of formatted I/O. For example, to write the array A(25,25), you should use an unformatted WRITE statement:

```
WRITE (7) A
```

This is much more efficient than a formatted WRITE statement. For example:

```
      WRITE (7,100) A
100   FORMAT (25(' ',25F5.21))
```

Using unformatted I/O has several benefits:

- It minimizes the CPU resources needed to perform the I/O operation because it avoids the translation to and from internal form.
- It increases the accuracy of floating-point data because it avoids a roundoff error on both input and output.
- It makes more efficient use of the capabilities of the I/O devices because it allows more data to be transmitted in a single operation.

11.4.5.2 Using the OPEN Statement's RECORDTYPE Keyword

For unformatted I/O, the default record type for the OPEN statement's RECORDTYPE keyword is 'SEGMENTED'. Each record is broken up into chunks called segments, and the I/O is performed on the segments rather than the records. This is not the most efficient way to perform unformatted I/O, but is used because it is the only way that works regardless of the size of the records.

You can often significantly improve the speed of the unformatted I/O by using the 'FIXED' or 'VARIABLE' record type.

- If the size of the records you are writing is always the same, you should always use the 'FIXED' record type. This allows the I/O subsystem to write exactly the data in your records; no space is wasted and no extra processing is needed.

- If your records are not all the same size, you can use the 'VARIABLE' record type and achieve some benefit over the use of 'SEGMENTED'. This record type requires the I/O subsystem to append a length word (which contains the number of bytes in the record) at the beginning of each record. It also requires the I/O subsystem to provide an additional level of buffering beyond that required for the 'FIXED' record type. This is better than using the 'SEGMENTED' type, which requires additional processing and buffering for each segment—as well as for each record.

11.4.5.3 Avoiding Run-Time Formats

When performing formatted I/O, two computation steps are required for each format string used.

1. The format string must be parsed to determine exactly what kind of formatting is required.
2. Each format code is matched with a corresponding data element from the I/O list of the program's I/O statement and the appropriate translation is performed.

When run-time formatting is used, both of these steps must be performed each time the I/O statement is executed. Run-time formatting is required when the format specifier in the I/O statement is the name of an array, an array element, or a nonconstant character expression.

The compiler avoids the first step of the formatting process when you do not use run-time formatting. It parses the FORMAT string at compile time and reduces it to a compact internal form that needs little processing at run-time to determine how each data item is to be translated.

You should minimize your use of run-time formats. In many cases, the use of variable format expressions can replace the use of run-time formats. Variable format expressions allow you to vary the exact format specification at run time while retaining the performance advantages of compiler preprocessing of FORMAT statements.

11.4.5.4 Avoiding the Use of the BACKSPACE Statement

The backspace operation is not directly supported on most VAX I/O devices (including magnetic tape drives) and therefore must be simulated by rereading the input file from the beginning. This simulation uses extra buffering to avoid rereading the input file with every BACKSPACE statement, but it is less efficient than using a direct access read when reading a disk file.

If a reread capability is required, it is more efficient to read the record into an internal file and read the internal file several times than to read and backspace to the record.

11.4.5.5 Using OPEN Statement Keywords to Control I/O

You can use the BLOCKSIZE and BUFFERCOUNT keywords in an OPEN statement in order to enhance the efficiency of I/O operations.

- *BLOCKSIZE*—One of the ways to reduce I/O overhead when you are using sequential access mode is to transfer larger blocks of data with each I/O operation. In this way, you can take advantage of the high data-transfer rates of the I/O devices, while minimizing the computational overhead of each I/O operation. In order to transfer larger blocks of data, use the BLOCKSIZE keyword when opening the file.
- *BUFFERCOUNT*—Often, you can improve the total execution time of a program by overlapping some of the computation with the I/O operations. The VAX Record Management Services (used by the VAX FORTRAN I/O system) does this automatically by using multiple buffers when it performs I/O operations. The use of multiple buffers allows the program to be processing one buffer while the I/O system is reading into, or writing from, another. You can control the use of multiple buffers in several ways, most easily by using the BUFFERCOUNT keyword when opening a file.

See the *VAX FORTRAN Language Reference Manual* for more information on the use of these keywords.

The following example shows the effect on the elapsed time of different values of BLOCKSIZE and BUFFERCOUNT on a typical VAX configuration. The example gives the relative elapsed CPU time for a program that writes and reads a 1000-element REAL array 100 times on a VAX-11/780.

The program has the following OPEN statement:

```
OPEN (UNIT=1, STATUS='NEW',
1     FORM='UNFORMATTED', RECORDTYPE='FIXED',
1     RECL=1000, BLOCKSIZE=IBLK, BUFFERCOUNT=IBUF)
```

The measures of relative elapsed CPU time achieved by this program (with BUFFERCOUNT settings of 1-4) are as follows:

BLOCKSIZE	**BUFFERCOUNT**			
	1	**2**	**3**	**4**
4000	1.00	1.00	.99	.99
8000	.69	.67	.67	.66
12000	.58	.54	.53	.53

As the example shows, you can reduce the elapsed time significantly by choosing the number and size of I/O buffers appropriately. However, because the optimal values are highly application dependent, you should experiment to determine which values produce the best results for any given program.

The OPEN and CLOSE statements provide explicit control over I/O devices and files, as shown in the following examples.

- The following statement allocates space for a file when the file is opened; this is a more efficient method than that of extending the size of the file dynamically.

  ```
  OPEN (UNIT=1, STATUS='NEW', INITIALSIZE=200)
  ```

- The following 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 (but requiring more memory).

  ```
  OPEN (UNIT=3, STATUS='NEW', BLOCKSIZE=8192)
  ```

- The following statement creates a file with implicit carriage control. Because the first character of each record is not used for carriage control, it can contain actual data.

  ```
   OPEN (UNIT=2, STATUS='NEW', FORM='FORMATTED',
  1      CARRIAGECONTROL='LIST')
  ```

11.4.5.6 Using Alternative I/O Methods

You can use alternative I/O methods when performance requirements call for faster I/O operations than can be achieved using standard FORTRAN I/O operations or when you need to use special features of the VAX I/O system that are not supported in the FORTRAN language. However, these alternative methods are not recommended unless you need them; their use makes your program nonstandard and thus degrades your ability to transport it to other computer systems.

The VAX/VMS I/O architecture has several levels, the highest level being the language support routines in the Run-Time Library. These routines in turn call the general I/O routines of the VMS Record Management Services (RMS) package, which are used for performing I/O by all DIGITAL-supported VAX native-mode language processors. RMS routines are described in the *VMS Record Management Services Manual.*

Often, you can access special features of RMS simply by using a USEROPEN routine. The USEROPEN keyword of the OPEN statement is used to specify the address of a user-written routine that will be called by the RTL I/O support routines instead of actually performing the RMS OPEN operation directly. Your USEROPEN routine can then modify the RMS control structures to access RMS features not directly supported by OPEN statement keywords. The use of the USEROPEN keyword is described in Chapter 7.

You can also call RMS I/O service routines directly from your FORTRAN program. To do this, you will need to become familiar with how to set up and manipulate the large data structures used by RMS in its service calls. These structures are called the File Access Block (FAB), Record Attributes Block (RAB), Extended Attributes Block (XAB), and File Name Descriptor Block (NAM). Refer to Section 6.5 for information on how to call VMS system service routines from FORTRAN programs. Refer to Chapter 7 for more information about calling RMS.

The lowest level of I/O services on the VMS system is the QIO level. This level specifies the actual operations that are to be performed by the peripheral devices. The QIO services are also callable directly from FORTRAN programs. These services also require data structures for specifying their arguments. Refer to the *VMS I/O User's Reference Manual: Part I* for more information on how to use the QIO services.

When using these alternative I/O methods, you should never try to use more than one level on the same I/O stream at the same time. For instance, do not mix standard FORTRAN read and write operations with direct calls to the RMS service routines for I/O on the same file at the same time. Such usage is not supported and cannot be made to work without a detailed understanding of every level involved. In addition, DIGITAL reserves the right to modify the interfaces used by the compiled code to the FORTRAN RTL I/O system, as well as the interfaces from these routines to RMS.

11.4.5.7 Implied-DO Loop Collapsing

In general, each I/O element in a FORTRAN I/O statement is processed by a separate call to the VAX FORTRAN Run-Time Library I/O processing routines. The computation overhead of these calls is most significant when using implied-DO loops. In the case of implied-DO loops, the compiler performs an optimization to reduce this overhead. In particular, for each innermost implied-DO loop in a nested group, it replaces the loop with a single call to an optimized Run-Time Library I/O routine that can transmit many I/O elements at once. This optimization is performed for both formatted and unformatted I/O, but is more effective in the case of unformatted I/O.

Behaviors of Formatted and Unformatted I/O

In formatted I/O, even though most calls to the Run-Time Library I/O support routines have been eliminated, the individual data items still require translating to or from external form. Thus, although the implied-DO loops are collapsed for formatted I/O, substantial processing is still needed for this type of I/O.

For unformatted I/O, on the other hand, collapsing the implied-DO loops often removes most of the processing required to perform the I/O. In particular, if the unformatted I/O transmits data items that are physically adjacent in memory, a level of buffering is eliminated and efficiency is substantially improved. For example:

```
DIMENSION A(200,300)
 . . .
READ (5) ((A(I,J),I=1,200),J=1,300)
```

This usage is particularly efficient because the compiler collapses the innermost loop and 200 elements at a time are read using a single block move operation. Note that efficiency is significantly degraded if the loops are nested in the opposite order. This is because block moves cannot be done, and locality of reference is destroyed because the natural order of subscript progression is not used.

Source Code Requirements for Optimization

Certain usages prevent the compiler from collapsing implied-DO loops and must be avoided if you intend to take advantage of this optimization.

When using formatted I/O, the implied-DO loop cannot be collapsed if the FORMAT statement contains a variable format expression or does not precede the I/O statement in the source program. This is necessary because the optimization is performed on the first phase of the compiler. Thus, it is impossible to test for the presence of a variable format expression in the FORMAT statement if it follows the I/O statement.

In addition, certain usages of the implied-DO loop control variables prevent collapsing in both formatted and unformatted I/O statements. In particular, collapsing cannot be done if the control variable is used as a dummy argument or in a COMMON statement, EQUIVALENCE statement, or VOLATILE statement. In addition, the control variable must be of INTEGER data type.

NOTE

The value of the loop control variable is unpredictable when the I/O statement terminates with an end of file or error condition. Your program should only reference the terminal value of this variable if the I/O statement completes normally.

11.4.5.8 Additional I/O Optimizations

You can often reduce the execution time of your FORTRAN programs by changing your programs to reflect the following considerations:

- Certain kinds of I/O lists can be optimized more effectively than others. For instance, an I/O list consisting of a single unformatted element (variable or array) does not have to be buffered in the Run-Time Library buffers. Also, implied-DO loops consisting of a single unnested element are transmitted as a single call to the Run-Time Library.
- 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 (for example, 32 bytes, 64 bytes, and so on). For relative organization files, RMS adds one overhead byte for fixed-length records and three overhead bytes for variable-length records, so the record length should be adjusted accordingly.

- If the approximate size of the file is known, it is more efficient to allocate disk space when a file is opened than to extend the file incrementally as records are written. You can make this allocation using the INITIALSIZE keyword in the OPEN statement.

11.5 Space Optimizations

Even though the VAX architecture allows the use of large memory spaces without overlays, it is still important to minimize the use of memory. Doing this often results in additional speed improvements and generally improves system throughput. The memory space optimizations performed by the compiler are described in the sections that follow. These fall into two categories: data size and code size.

11.5.1 Data Size Optimizations

The compiler optimizes the use of data space by avoiding duplication of quantities requiring space and by eliminating data items that are not actually used by the program.

11.5.1.1 Constant Pooling

Only one copy of a given constant value is ever allocated memory space. If that constant value is used in several places in the program, all references point to that value.

11.5.1.2 Argument List Merging

Argument-list data structures are built by the compiler to describe actual argument lists used by your program. Only one copy of a given actual argument list is built, even if it is used by more than one CALL or FUNCTION reference. For example:

```
XYZ = A + FUNC(B,C)
 . . .
CALL SUBA(B,C)
 . . .
```

The argument list (B,C) is allocated only once in memory.

11.5.1.3 Dead Variable Elimination

Variables whose uses have all been removed by value propagation optimizations, or by register usages, are not allocated in memory. In the memory map section of the compiler output listing, such variables appear with double asterisks (**) for their memory locations.

11.5.2 Code Size Optimizations

The compiler performs several optimizations that reduce the amount of memory required by the object code. These optimizations also frequently reduce execution time.

11.5.2.1 Local Storage Allocation

The allocation of local variables and arrays declared in your program (that is, those variables and arrays not in common blocks or dummy arguments) is chosen so as to minimize the offsets from the base register used to address them. The VAX instruction modes allow offsets of different sizes (0, 1, 2, or 4 bytes) for exactly this purpose. A considerable execution speed advantage can often be achieved by using the smallest necessary offset size.

11.5.2.2 Jump Branch Resolution

In addition to allowing different sizes of operand offsets, the VAX instruction set includes a variety of branch-type instructions for controlling the flow of program execution. These different branch instructions allow different offset sizes for code space optimization. The compiler optimizes the use of these instructions in order to choose the smallest offsets required for each branch in the program.

11.5.2.3 Dead Code Eliminations

The compiler can sometimes detect that some parts of the program will never be executed. In this case, it removes the code entirely. It does this in several different ways and in several phases of the compiler. These dead code eliminations do not necessarily mean that poor programming practice has been followed; often, it is simply the result of using PARAMETER constants. For this reason, no warning messages are given when dead code is eliminated.

Code Reordering for Branch Elimination

The compiler reduces the number of branch type instructions in the program by arranging the order in which the statements appear in the machine code. This optimization effectively removes unneeded GOTO operations from the machine code.

Elimination of Unreachable Code

Source code that can never actually be executed is said to be dead code. For example:

```
IF (.FALSE.) A = B
```

In this case, the assignment statement will never be executed.

Even though the existence of dead code in a program does not generally affect execution speed, it does occupy memory space needlessly. It also causes larger offsets to be needed in branch instructions that span the dead code. The compiler detects dead code in two ways and always eliminates it from the object program.

- *Lexically detected - warning issued.* The compiler detects dead code first by identifying source statements that cannot be reached. For example:

  ```
   . . .
  GO TO 100
  I = J*K
   . . .
  ```

  The assignment statement in this example can never be executed because the GOTO statement is unconditional and because the assignment statement has no label that can be referenced by a control statement, such as DO or GOTO. When the compiler detects such usage, it issues a warning message indicating that the statement cannot be reached.
- *Detected by value propagation - no warning issued.* Sometimes, it is not apparent from the source program that the dead code cannot be reached. For example:

  ```
  I = 100
   . . .
  IF (I .LT. 0) A = B*C
   . . .
  ```

  In this example, the assignment can never be executed because the value of I will never be less than 0. When the compiler detects such usage, it does not issue a warning that the code cannot be reached.

Elimination of Operations on Dead Variables

The compiler analyzes the use of selected variables to determine if they have any effect on the output of the program unit. If they do not, they are called dead variables. The compiler optimizes both speed and space by eliminating all operations on dead variables.

11.6 Compiler Optimization Example

The example in this section shows many of the optimization techniques used by the VAX FORTRAN compiler. The first part (Example 11–1) shows a complete FORTRAN subroutine, a relaxation function often used in engineering applications. This subroutine is a two-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 (Example 11–2) shows the VAX machine code generated by the FORTRAN compiler. Several compiler optimizations are indicated by the circled numbers next to the generated code lines. These are described in the notes that follow the example.

Example 11–1: RELAX Source Program

```
0001              SUBROUTINE RELAX2(EPS)
0002
0003              PARAMETER (M=40, N=60)
0004              DIMENSION X(0:M,0:N)
0005              COMMON X
0006
0007              LOGICAL DONE
0008
0009     1        DONE = .TRUE.
0010
```

Example 11–1 Cont'd. on next page

Example 11–1 (Cont.): RELAX Source Program

```
0011            DO 10 J=1,N-1
0012            DO 10 I=1,M-1
0013                XNEW = (X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1))/4
0014                IF (ABS(XNEW-X(I,J)) .GT. EPS) DONE = .FALSE.
0015    10          X(I,J) = XNEW
0016
0017            IF (.NOT. DONE) GO TO 1
0018
0019            RETURN
0020            END
```

Example 11–2: RELAX Machine Code (Optimized)

```
              .TITLE  RELAX2
              .IDENT  01

0000          .PSECT  $BLANK
0000   X:

0000          .PSECT  $CODE
                                                        ; 0001
0000   RELAX2::
0000          .WORD   ^M<IV,R2,R3,R4,R6,R7>
                                          ❶           ; 0009
0002          NOP                         ❷
0003          NOP
0004   .1:
0004          MNEGL   #1, R0              ❸
                                                        ; 0011
0007          MOVL    #1, R1              ❹
000A          NOP                         ❷
000B          NOP
000C   L$1:
```

Example 11–2 Cont'd. on next page

Example 11–2 (Cont.): RELAX Machine Code (Optimized)

```
                                                                 ; 0012
000C          MOVL    #1, R2                         ❺
000F          MULL3   #41, R1, R3                    ❻
0013          MOVAF   X[R3], R4                      ❼
001B          NOP                                    ❷
001C  L$2:
                                                                 ; 0013
001C          ADDF3   8(R4), (R4)+, R6               ❽
0021          ADDF2   -164(R4), R6                   ❾
0026          ADDF2   164(R4), R6
002B          MULF2   #^X3F80, R6                    ❿ ⓫
                                                                 ; 0014
0032          SUBF3   (R4), R6, R7
0036          BICW2   #^X8000, R7                    ⓬
003B          CMPF    R7, @EPS(AP)                   ⓭
003F          BLEQ    L$3
0041          CLRL    RO
0043  L$3:
                                                                 ; 0015
0043          MOVL    R6, (R4)
0046          AOBLEQ  #39, R2, L$2                   ⓮
004A          AOBLEQ  #59, R1, L$1
                                                                 ; 0017
004E          BLBC    RO, .1                         ⓯
                                                                 ; 0019
0051          RET                                    ⓰
              .END
```

Notes to Example 11–2

❶ All local variables are eliminated, so no $LOCAL PSECT is needed (and no base register is needed to point to it).

❷ Loop tops are aligned on longword boundaries using NOP instructions.

❸ Register assignment for DONE; short form of constant.

❹ Register assignment for J.

❺ Register assignment for I.

❻ Common subexpression (J*41) is hoisted from loop and assigned to a register.

❼ Base address (X(1,1)) is loaded into a register. Six references to it.

❽ Autoincrement address mode for X(I–1,J).

❾ Register 6 is used for all temporary variables for line 7.

⑩ Peephole optimization; a divide by 4.0 is replaced by a multiply by 0.25.

⑪ XNEW loaded into Register 6, allowing for two-operand multiply.

⑫ In-line ABS function.

⑬ Flow Boolean optimization for IF statement.

⑭ DO loop control using a single AOBLEQ (add one and branch less than or equal) instruction.

⑮ Logical test and branch in a single instruction.

⑯ Only 82 code bytes total (25 less than VAX FORTRAN, Version 3).

Chapter 12

Using Structures and Records

VAX FORTRAN structures and records allow you to group associated data items together in a common entity declaration instead of handling them in separate variable and array declarations.

Like arrays, records can contain one or more data elements. Unlike arrays, however, the data elements in a record, called fields, can have different data types. Also, each field of a record has a unique name that identifies it; whereas, each element of an array is identified by a unique numeric index.

This chapter provides an overview of how records can be used in VAX FORTRAN programs. Detailed information about the specifics of record use is provided in the *VAX FORTRAN Language Reference Manual.* Topics addressed in that manual include the following:

- The way to reference records and how records appear in memory
- The use of records in assignment statements
- The format of the RECORD statement
- The format of structure declaration blocks, which define the fields or groups of fields within a record

12.1 Structures

In VAX FORTRAN, structures are used to describe the form of records. You can think of these structures as templates that define the form and size of records.

You define a structure with a block of statements, beginning with a STRUCTURE statement and ending with an END STRUCTURE statement. You can use the following statements within a structure declaration block:

- Statements that appear very much like data type declaration statements. These statements form structure declaration blocks and describe the fields contained within the structure.
- Statements that define substructures (nested structure declarations and RECORD statements) and mapped common areas (union declarations). These constructs are not discussed in this chapter; see the *VAX FORTRAN Language Reference Manual* for details.
- PARAMETER statements. A PARAMETER statement in a structure declaration block has its normal effect of assigning a symbolic name to a constant.

You specify the name of a structure in the STRUCTURE statement. The RECORD statement uses this name to identify the structure that is to be made into a record (or structured field).

The following example defines the structure DATE. It contains three fields: DAY, MONTH, and YEAR. Note that the field YEAR is initialized with 1986. Any records defined to have the structure DATE will have their YEAR field initialized to 1986.

```
STRUCTURE /DATE/
    LOGICAL*1 DAY, MONTH
    INTEGER*2 YEAR /1986/
END STRUCTURE
```

The following example defines the structure PERSON, which might be used to hold information about an individual. It contains the fields NAME, SEX, and BIRTH_DATE. Note that the fields NAME and BIRTH_DATE are themselves structured; that is, they are substructures within the structure PERSON. NAME's structure declaration (unnamed) contains the fields LAST_NAME, FIRST_NAME, and MIDDLE_INITIAL. BIRTH_DATE has the structure of DATE, the structure defined in the preceding example.

```
STRUCTURE /PERSON/
    STRUCTURE NAME
        CHARACTER*20 LAST_NAME, FIRST_NAME
        CHARACTER*1  MIDDLE_INITIAL
    END STRUCTURE
    LOGICAL*1    SEX
    RECORD /DATE/ BIRTH_DATE
END STRUCTURE
```

See the *VAX FORTRAN Language Reference Manual* for detailed information about structure declarations and their syntactical elements.

12.2 Records

You use the RECORD statement to define record scalars and arrays, in much the same way that you use type declaration statements. For example, the following RECORD statement, based on the structure PERSON shown in the preceding section, could be used:

```
RECORD /PERSON/ FATHER, MOTHER, CHILDREN(10)
```

The preceding statement creates twelve records with the structure PERSON. In each of the twelve records, all fields are initially undefined, with the exception of BIRTH_DATE.YEAR, which has been initialized to 1986 in the structure declaration DATE.

See the *VAX FORTRAN Language Reference Manual* for information about how to reference records and fields and about how they appear in memory.

12.3 Uses of Records

When you have several data arrays, each containing a different, though related, type of information, the use of records allows you to use the same index to refer to each array.

As an example, consider a FORTRAN program that maintains and manipulates a symbol table. The symbol table consists of three arrays: the first contains the symbol names, the second contains the symbol values, and the third contains a flag signaling whether the symbol is defined. As an example, the declaration in FORTRAN-77 could be as follows:

```
PARAMETER (MAXSYM=1000)
CHARACTER*16 SYMBOL_NAME(MAXSYM)
INTEGER*4    SYMBOL_VALUE(MAXSYM)
LOGICAL*1    SYMBOL_FLAG(MAXSYM)
```

Note that each array is declared separately and that the data items, although related, are declared (and later manipulated) separately. For example, to read or write such related information from or to a file, you must specify each piece individually, as in the following WRITE statement:

```
WRITE (10) SYMBOL_NAME(I), SYMBOL_VALUE(I), SYMBOL_FLAG(I)
```

With structures and records, however, the definition allows you to group the related information together and to refer to that information as a whole in many cases. Thus, the symbol table declaration could appear as follows:

```
STRUCTURE /SYM/
    CHARACTER*16   NAME
    INTEGER*4      VALUE
    LOGICAL*1      FLAG
END STRUCTURE
 . . .
RECORD /SYM/ SYMBOL(MAXSYM)
```

These declarations create only one array, the record array SYMBOL. Each element of SYMBOL has the form (structure) of SYM. This means that each element of SYMBOL consists of the three fields NAME, VALUE, and FLAG. Note that the related information about an individual symbol—its name, value, and flag—is now one element of a record array. As a result, you can refer to a symbol table (that is, SYMBOL instead of individual arrays such as SYMBOL_NAME), a single symbol I (for example, SYMBOL(I) instead of SYMBOL_NAME(I)), or any of the fields in symbol I (for example, SYMBOL(I).NAME). Thus, the previous WRITE statement would be changed as follows:

```
WRITE (10) SYMBOL(I)
```

This statement is equivalent to the following statement:

```
WRITE (10) SYMBOL(I).NAME, SYMBOL(I).VALUE, SYMBOL(I).FLAG
```

Without records, VAX FORTRAN programs would have to use COMMON blocks in some cases—such as with arguments of system service calls—to pass structured information to subroutines (see Section 6.5.4.4 for information on data structure arguments). Such routines expect to receive the address of either a list, control block, or vector, and the COMMON statement constructs these arguments, with no empty spaces between adjacent items, in order of declaration. The resulting COMMON blocks can be used as records but do not have the flexibility of records.

For example, a call to the SYS$GETJPI system service requires the address of a sequence of items consisting of two words followed by two longwords. With records, this call can be achieved with the following code:

```
STRUCTURE /GETJPI_ITEM/
    INTEGER*2  W_LEN, W_CODE
    INTEGER*4  L_ADDR, L_LENADDR/0/
END STRUCTURE
RECORD /GETJPI_ITEM/ GETJPIARG(5)
...
GETJPIARG(4).W_LEN = 4
GETJPIARG(4).W_CODE = JPI$_CPUTIM
GETJPIARG(4).L_ADDR = %LOC(LCL_VALUES(4))
...
CALL SYS$GETJPI(,,,GETJPIARG,,,)
```

As this example shows, the primary advantage to using records is that they enable you to group conceptually related data together in one entity—regardless of conflicts in data types.

Chapter 13

Using Character Data

VAX FORTRAN's 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 concatenation operator (//) can be used to form a single character string from two or more separate strings.

This chapter provides information on the following topics:

- Character substrings (Section 13.1)
- Building character strings (Section 13.2)
- Character constants (Section 13.3)
- Declaring character data (Section 13.4)
- Initializing character variables (Section 13.5)
- Passed-length character arguments (Section 13.6)
- Character library functions (Section 13.7)
- Character input/output (Section 13.8)

Section 13.9 provides an example of how character data can be used in VAX FORTRAN programs.

13.1 Character Substrings

You can select certain segments (substrings) from a character variable, character array element, or character record field by specifying the name of the variable, array element, or record field, followed by delimiter values indicating the positions of the leftmost and rightmost characters in the substring. For example, you could define the character variable NAME to contain the following string:

```
ROBERTΔWILLIAMΔBOBΔJACKSON
```

To extract the substring BOB from the variable NAME, 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, you could specify the variable as follows:

```
NAME(:18)
```

This results in the following substring:

```
ROBERTΔWILLIAMΔBOB
```

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

```
NAME(16:)
```

This results in the following substring:

```
BOBΔJACKSON
```

13.2 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 want to create a variable called NAME, consisting of the values of the variables FIRSTNAME, MIDDLENAME, NICKNAME, and 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 variables contained the following values:

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

The values would be stored individually as follows:

```
ROBERTΔΔΔΔΔΔ
WILLIAMΔΔΔΔΔ
BOBΔΔΔ
JACKSONΔΔΔΔΔ
```

Then, when concatenated and stored in NAME, they form the following string:

```
ROBERTΔΔΔΔΔΔWILLIAMΔΔΔΔΔBOBΔΔΔJACKSONΔΔΔΔΔ
```

Applying the substring extraction facility described in Section 13.1, you can get the stored nickname by specifying the variable as follows:

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

This picks up the 6-character substring BOBΔΔΔ (including trailing blanks) in the variable NAME.

13.3 Character Constants

Character constants are strings of characters enclosed in apostrophes. You assign a character value to a character variable in much the same way you 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 is truncated on the right. For example, the following source statements produce a result of AB:

```
CHARACTER*2 XYZ

XYZ = 'ABC'
```

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'
```

This results in the following string being stored in XYZ:

```
ABCΔΔΔ
```

The previous contents of XYZ are overwritten. Thus, if the previous contents of XYZ were DEFGHI, the result would still be ABCΔΔΔ.

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

```
CHARACTER*(*) TITLE
PARAMETER (TITLE = 'THE METAMORPHOSIS')
```

This PARAMETER statement example assigns the symbolic name TITLE to the character constant THE METAMORPHOSIS.

You can use the symbolic name TITLE anywhere a character constant is allowed.

To include an apostrophe as part of the constant, specify two consecutive apostrophes. For example:

```
CHARACTER*(*) TITLE
PARAMETER (TITLE = 'MARTHA''S VINEYARD')
```

This results in the character constant MARTHA'S VINEYARD.

The value assigned to a character parameter can be any compile-time constant character expression. Note in particular that the CHAR intrinsic function (see Section 13.7.1) with a constant argument is a compile-time constant expression; therefore, you can assign nonprinting characters to parameter constants. For example:

```
CHARACTER*(*) CRLF
PARAMETER (CRLF=CHAR(13)//CHAR(10))
```

13.4 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 as 6-byte variables, while AGE is defined as a 2-byte variable.

13.5 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',
1    'DOE','BROWN','GREEN'/
```

Note that NAME contains 10 blanks, while each array element in TEAM contains 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.

You can also initialize character variables within the character declaration, as shown in the following example:

```
CHARACTER*10 METALS(3)/'LEAD','IRON','GOLD'/
```

13.6 Passed-Length Character Arguments

In writing subprograms that manipulate character data, you can get the subprogram to accept actual character 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 is 20; in the second call, the length of S is 53.

You can use the CHARACTER*(*) notation to define the length of parameter character constants. The actual length is then the length of the character constant that is assigned to the parameter name in a PARAMETER statement.

The FORTRAN function LEN can be used to determine the actual length of the string (see Section 13.7.4).

13.7 Character Intrinsic Functions

VAX FORTRAN supports the following character intrinsic functions:

- CHAR
- ICHAR
- INDEX
- LEN
- LGE, LGT, LLE, LLT

The following sections describe these functions.

13.7.1 CHAR Intrinsic Function

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

```
CHAR(i)
```

The notation i represents an integer expression equivalent to an ASCII code.

13.7.2 ICHAR Intrinsic Function

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

```
ICHAR(c)
```

The notation c represents 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.

13.7.3 INDEX Intrinsic Function

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

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

The notations c1 and c2 represent character expressions. Character expression c1 specifies the string to be searched for a match with the value of the substring specified in character expression c2.

If the INDEX function 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 DOGCATFISHCAT, the return value of the INDEX function is 4.

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

If there are multiple occurrences of the substring, INDEX locates the first (leftmost) one. Use of the INDEX function is illustrated in Example 13–1.

13.7.4 LEN Intrinsic Function

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

```
LEN(c)
```

The notation c represents a character expression.

13.7.5 LGE, LGT, LLE, LLT Intrinsic Functions

The lexical comparison functions LGE, LGT, LLE, and LLT are defined by the FORTRAN–77 standard to make comparisons between two character expressions using the ASCII collating sequence. The result is the logical value .TRUE. if the lexical relation is true, and .FALSE. if the lexical relation is not true. The functions have the forms:

```
LGE(c1,c2)
LGT(c1,c2)
LLE(c1,c2)
LLT(c1,c2)
```

The notations c1 and c2 represent character expressions.

You may want to use these functions in FORTRAN programs that may be used on computers that do not support the ASCII character set. In VAX FORTRAN, the lexical comparison functions are equivalent to the .GE., .GT., .LE., .LT. relational operators. For example, the following statements are equivalent:

```
IF (LLE(string1, string2)) GO TO 100
IF (string1 .LE. string2) GO TO 100
```

13.8 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 65535 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 one-dimensional array comprising six 4-byte elements is filled with 24 characters from logical unit 12.

13.9 Character Data Examples

Example 13–1 is a program that uses the VAX FORTRAN character data type to manipulate the letters of the alphabet.

Example 13–1: Character Data Program Example

```
      CHARACTER C, ALPHABET*26

      DATA ALPHABET/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/

      WRITE (6,90)
90    FORMAT (' CHARACTER EXAMPLE PROGRAM OUTPUT'/)

      DO I=1,26
          WRITE (6,*) ALPHABET
          ALPHABET = ALPHABET(2:)//ALPHABET(1:1)
      END DO

      CALL REVERSE(ALPHABET)
      WRITE (6,*) ALPHABET

      CALL REVERSE(ALPHABET(1:13))
      WRITE (6,*) ALPHABET

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

      WRITE (6,*) 'END OF CHARACTER EXAMPLE PROGRAM'
      END

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

      J = LEN(S)
      DO I=1,J/2
          T = S(I:I)
          S(I:I) = S(J:J)
          S(J:J) = T
          J = J - 1
      END DO
      END

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

      I = 1
      MARKS = ' '
```

Example 13–1 Cont'd. on next page

Example 13–1 (Cont.): Character Data Program Example

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

      WRITE (6,91) S, MARKS
91    FORMAT (2(/1X,A))
      END
```

The program in Example 13–1 produces the following output:

```
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
```

Chapter 14
Using Indexed Files

Traditionally, sequential and direct access have been the only file access modes available to FORTRAN programs. To overcome some of the limitations of these access modes, VAX FORTRAN supports a third access mode, called keyed access, which allows you to retrieve records, at random or in sequence, based on key fields that are established when you create a file with indexed organization. (See Section 4.2.3.4 for details about keyed access mode.)

You can access files with indexed organization using sequential access or keyed access, or a combination of both.

- Keyed access retrieves records randomly based on the particular key fields and key values that you specify.
- Sequential access retrieves records in a sequence based on the direction of the key and on the values within the particular key field that you specify.

The combination of keyed and sequential access is commonly referred to as the Indexed Sequential Access Method (ISAM). Once you have read a record by means of an indexed read request, you can then use a sequential read request to retrieve records with ascending key field values, beginning with the key field value in the record retrieved by the initial read request.

Indexed organization is especially suitable for maintaining complex files in which you want to select records based on one of several criteria. For example, a mail-order firm could use an indexed organization file to store its customer list. Key fields could be a unique customer order number, the customer's zip code, and the item ordered. Reading sequentially based on the zip-code key field would enable you to produce a mailing list sorted by zip code. A similar operation based on customer-order-number key field or item-number key field would enable you to list the records in sequences of customer order numbers or item numbers.

This chapter provides information of the following topics:

- Creating an indexed file (Section 14.1)
- Writing records to an indexed file (Section 14.2)
- Reading records from an indexed file (Section 14.3)
- Deleting records from an indexed file (Section 14.5)
- Updating records in an indexed file (Section 14.4)

In addition, information is provided about the effects of read and write operations on positioning your program to records within an indexed file (Section 14.6) and about how to build logic into your programs to handle exception conditions that commonly occur (Section 14.7).

14.1 Creating an Indexed File

You can create a file with an indexed organization by using either the FORTRAN OPEN statement or the RMS EDIT/FDL Utility.

- Use the OPEN statement to specify the file options supported by FORTRAN.
- Use the EDIT/FDL Utility to select features not directly supported by FORTRAN.

Any indexed file created with EDIT/FDL can be accessed by FORTRAN I/O statements.

When you create an indexed file, you define certain fields within each record as key fields. One of these key fields, called the *primary key,* is identified as key number zero and must be present in every record. Additional keys, called *alternate keys,* can also be defined; they are numbered from 1 through a maximum of 254. An indexed file can have as many as 255 key fields defined. In practice, however, few applications require more than 3 or 4 key fields.

The data types used for key fields must be either INTEGER*2, INTEGER*4, or CHARACTER.

In designing an indexed file, you must decide the byte positions of the key fields. For example, in creating an indexed file for use by a mail-order firm, you might define a file record to consist of the following fields:

```
STRUCTURE /FILE_REC_STRUCT/
   INTEGER*4 ORDER_NUMBER       ! Positions 1:4, key 0
   CHARACTER*20 NAME            ! Positions 5:24
   CHARACTER*20 ADDRESS         ! Positions 25:44
   CHARACTER*19 CITY            ! Positions 45:63
   CHARACTER*2 STATE            ! Positions 64:65
   CHARACTER*9 ZIP_CODE         ! Positions 66:74, key 1
   INTEGER*2 ITEM_NUMBER        ! Positions 75:76, key 2
END STRUCTURE
.
.
.
RECORD /FILE_REC_STRUCT/ FILE_REC
```

Given this record definition, you could use the following OPEN statement to create an indexed file:

```
OPEN (UNIT=10, FILE='CUSTOMERS.DAT', STATUS='NEW',
1     ORGANIZATION='INDEXED', ACCESS='KEYED',
2     RECORDTYPE='VARIABLE', FORM='UNFORMATTED',
3     RECL=19,
4     KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER),
5     IOSTAT=IOS, ERR=9999)
```

This OPEN statement establishes the attributes of the file, including the definition of a primary key and two alternate keys. Note that the definitions of the integer keys do not explicitly state INTEGER*4 and INTEGER*2. The data type sizes are determined by the number of character positions allotted to the key fields, which in this case are 4 and 2 character positions, respectively.

If you specify the KEY keyword when opening an existing file, the key specification that you give must match that of the file.

VAX FORTRAN uses RMS default key attributes when creating an indexed file. These defaults are as follows:

- The values in primary key fields cannot be changed when a record is rewritten. Duplicate values in primary key fields is prohibited.
- The values in alternate key fields can be changed. Duplicate values in alternate key fields is permitted.

You can use the EDIT/FDL Utility or a USEROPEN routine to override these defaults and to specify other values not supported by VAX FORTRAN, such as null key field values, null key names, and key data types other than integer and character.

Refer to the *VAX FORTRAN Language Reference Manual* for information on the use of the USEROPEN keyword in OPEN statements. The *VMS Record Management Services Manual* provides additional information on indexed file options.

Use of the EDIT/FDL Utility is explained in detail in the *VMS File Definition Language Facility Manual* and the *Guide to VMS File Applications.*

14.2 Writing Indexed Files

You can write records to an indexed file with either formatted or unformatted indexed WRITE statements. Each write operation inserts a new record into the file and updates the key indexes so that the new record can be retrieved in a sequential order based on the values in the respective key fields.

For example, you could add a new record to the file for the mail-order firm (see Section 14.1) with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

The next sections discuss the following topics:

- The effects of writing records with duplicate values in key fields
- The method by which you can prevent an alternate key field in a record from being indexed during a write operation

14.2.1 Duplicate Values in Key Fields

It is possible to write two or more records with the same value in a single key field. The attributes specified for the file when it was created determine whether this duplication is allowed. By default, VAX FORTRAN creates files that allow duplicate alternate key field values and prohibit duplicate primary key field values. If duplicate key field values are present in a file, the records with equal values are retrieved on a first-in/first-out basis.

For example, assume that five records are written to an indexed file in this order (for clarity, only key fields are shown):

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375
942	02163	2736
903	14853	375
1348	44901	1047
1263	33032	690

If the file is later opened and read sequentially by primary key (ORDER_NUMBER), the order in which the records are retrieved is not affected by the duplicated value (375) in the ITEM_NUMBER key field. In this case, the records would be retrieved in the following order:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
903	14853	375
942	02163	2736
1023	70856	375
1263	33032	690
1348	44901	1047

However, if the read operation is based on the second alternate key (ITEM_NUMBER), the order in which the records are retrieved is affected by the duplicate key field value. In this case, the records would be retrieved in the following order:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375
903	14853	375
1263	33032	690
1348	44901	1047
942	02163	2736

Notice that the records containing the same key field value (375) are retrieved in the order in which they were written to the file.

14.2.2 Preventing the Indexing of Alternate Key Fields

When writing to an indexed file that contains variable-length records, you can prevent entries from being added to the key indexes for any alternate key fields. This is done by omitting the names of the alternate key fields from the WRITE statement. The omitted alternate key fields must be at the end of the record; another key field cannot be specified after the omitted key field.

For example, the last record (ORDER_NUMBER 1263) in the mail-order example could be written with the following statement:

```
WRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC.ORDER_NUMBER,
1       FILE_REC.NAME, FILE_REC.ADDRESS, FILE_REC.CITY,
1       FILE_REC.STATE, FILE_REC.ZIP_CODE
```

Because the field name FILE_REC.ITEM_NUMBER is omitted from the WRITE statement, an entry for that key field is not created in the index. As a result, an attempt to read the file using the alternate key ITEM_NUMBER would not retrieve the last record and would produce the following listing:

ORDER_NUMBER	ZIP_CODE	ITEM_NUMBER
1023	70856	375
903	14853	375
1348	44901	1047
942	02163	2736

You can omit only trailing alternate keys from a record; the primary key must always be present.

14.3 Reading Indexed Files

You can read records in an indexed file with either sequential or indexed READ statements (formatted or unformatted) under the keyed mode of access. By specifying ACCESS='KEYED' in the OPEN statement, you enable both sequential and keyed access to the indexed file.

Indexed READ statements position the file pointers (see Section 14.6) at a particular record, determined by the key field value, the key-of-reference, and the match criterion. Once you retrieve a particular record by an indexed READ statement, you can then use sequential access READ statements to retrieve records with increasing key field values.

The form of the external record's key field must match the form of the value you specify in the KEY keyword. Thus, if the key field contains character data, you should specify the KEY keyword value as a CHARACTER data type. If the key field contains binary data, then the KEY keyword value should be of INTEGER data type.

Note that if you write a record to an indexed file with formatted I/O, the data type is converted from its internal representation to an external representation. As a result, the key value must be specified in the external form when you read the data back with an indexed read. Otherwise, a match will occur when you do not expect it.

The following FORTRAN program segment prints the order number and zip code of each record where the first five characters of the zip code are greater than or equal to '10000' but less than '50000':

```
C    Read first record with ZIP_CODE key greater than or
C    equal to '10000'.

        READ (UNIT=10,KEYGE='10000',KEYID=1,IOSTAT=IOS,ERR=9999)
        1     FILE_REC

C    While the zip code previously read is within range, print
C    the order number and zip code, then read the next record.

        DO WHILE (FILE_REC.ZIP_CODE .LT. '50000')
          PRINT *, 'Order number', FILE_REC.ORDER_NUMBER, 'has zip code',
        1         FILE_REC.ZIP_CODE
          READ (UNIT=10,IOSTAT=IOS,END=200,ERR=9999)
        1       FILE_REC

C    END= branch will be taken if there are no more records
C    in the file.

          END DO
200       CONTINUE
```

The error branch on the keyed READ in this example is taken if no record is found with a zip code greater than or equal to '10000'; an attempt to access a nonexistent record is an error. If the sequential READ has accessed all records in the file, however, an end-of-file status occurs, just as with other file organizations.

If you want to detect a failure of the keyed READ, you can examine the I/O status variable, IOS, for the appropriate error number (see Table 5–1 for a list of the returned error codes).

14.4 Updating Records

The REWRITE statement updates existing records in an indexed file. You cannot replace an existing record simply by writing it again; a WRITE statement would attempt to add a new record.

An update operation is accomplished in two steps. First, you must read the record in order to make it the current record. Next, you execute the REWRITE statement. For example, to update the record containing ORDER_NUMBER 903 (see prior examples) so that the NAME field becomes 'Theodore Zinck', you might use the following FORTRAN code segment:

```
READ (UNIT=10,KEY=903,KEYID=0,IOSTAT=IOS,ERR=9999) FILE_REC
FILE_REC.NAME = 'Theodore Zinck'
REWRITE (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
```

When you rewrite a record, key fields may change. The attributes specified for the file when it was created determine whether this type of change is permitted.

14.5 Deleting Records

To delete records from an indexed file, you use the DELETE statement. The DELETE and REWRITE statements are similar; a record must first be locked by a READ statement before it can be operated on.

The following FORTRAN code segment deletes the second record in the file with ITEM_NUMBER 375 (refer to previous examples):

```
READ (UNIT=10,KEY=375,KEYID=2,IOSTAT=IOS,ERR=9999)
READ (UNIT=10,IOSTAT=IOS,ERR=9999) FILE_REC
IF (FILE_REC.ITEM_NUMBER .EQ. 375) THEN
    DELETE (UNIT=10, IOSTAT=IOS, ERR=9999)
ELSE
    PRINT *, 'There is no second record.'
END IF
```

Deletion removes a record from all defined indexes in the file.

14.6 Current Record and Next Record Pointers

The RMS file system maintains two pointers into an open indexed file: the *next record* pointer and the *current record* pointer.

- The next record pointer indicates the record to be retrieved by a sequential read. When you open an indexed file, the next record pointer indicates the record with the lowest primary key field value. Subsequent sequential read operations cause the next record pointer to be the one with the next higher value in the same key field. In case of duplicate key field values, records are retrieved in the order in which they were written.
- The current record pointer indicates the record most recently retrieved by a READ operation; it is the record that is locked from access by other programs sharing the file. The current record is the one operated on by the REWRITE statement and the DELETE statement. The current record is undefined until a read operation is performed on the file. Any file operation other than a read causes the current record pointer to become undefined. Also, an error results if a rewrite or delete operation is performed when the current record pointer is undefined.

14.7 Exception Conditions

You can expect to encounter certain exception conditions when using indexed files. The two most common of these conditions involve valid attempts to read locked records and invalid attempts to create duplicate keys. Provisions for handling both of these situations should be included in a well-written program.

When an indexed file is shared by several users, any read operation may result in a "specified record locked" error. One way to recover from this error condition is to ask if the user would like to reattempt the read. If the user's response is positive, then the program can go back to the READ statement. For example:

```
     INCLUDE '($FORIOSDEF)'

100  READ (UNIT=10,IOSTAT=IOS) DATA

     IF (IOS .EQ. FOR$IOS_SPERECLOC) THEN
         TYPE *, 'That record is locked. Press RETURN'
         TYPE *, 'to try again, or CONTROL_Z to discontinue'
         READ (UNIT=*,FMT=*,END=900)
         GO TO 100
     ELSE IF (IOS .NE. 0) THEN
         CALL ERROR (IOS)
     END IF
```

You should avoid looping back to the READ statement without first providing some type of delay (caused by a request to try again, or to discontinue, as in this example). If your program reads a record but does not intend to modify the record, you should place an UNLOCK statement immediately after the READ statement. This technique reduces the time that a record is locked and permits other programs to access the record.

The second exception condition, creation of duplicate keys, occurs when your program tries to create a record with a key field value that is already in use. When duplicate key field values are not desirable, you might have your program prompt for a new key field value whenever an attempt is made to create a duplicate. For example:

```
     INCLUDE '($FORIOSDEF)'

200  WRITE (UNIT=10,IOSTAT=IOS) KEY_VAL, DATA

     IF    (IOS .EQ. FOR$IOS_INCKEYCHG) THEN
            TYPE *, 'This key field value already exists. Please enter'
            TYPE *, 'a different key field value, or press CONTROL_Z'
            TYPE *, 'to discontinue this operation.'
            READ (UNIT=*,FMT=300,END=999) KEY_VAL
            GO TO 200
     ELSE IF (IOS .NE. 0) THEN
            CALL ERROR (IOS)
     END IF
```

Chapter 15

VAX FORTRAN Support for Parallel Processing

This chapter describes how to achieve parallel processing using features incorporated in the VAX FORTRAN programming language.

This chapter provides the following information:

- An overview of VAX FORTRAN support for parallel processing (Section 15.1)
- A review of how to prepare VAX FORTRAN parallel applications (Section 15.2)
- A description of the data dependence problems associated with parallel processing and the methods you can use to resolve some of these problems (Section 15.3)
- A description of system and user account parameters that you may need to tune for parallel processing (Section 15.4)
- A discussion about how to debug a VAX FORTRAN program executing in parallel (Section 15.5)
- An example of a VAX FORTRAN program that can be run in parallel and some sample transformations of DO loops containing linear recurrences (Section 15.6)
- Detailed descriptions of the /PARALLEL qualifier for the FORTRAN command line, VAX FORTRAN compiler directive statements, logical names used by the VAX FORTRAN Run-Time Library, and an intrinsic function for use in VAX FORTRAN parallel-processing applications (Section 15.7)

15.1 Overview of VAX FORTRAN Parallel Processing

Parallel processing entails executing segments of a program *concurrently* on two or more processors in a multiprocessing system (for example, a VAX 8300 or VAX 8800; not to be confused with a VAXcluster system). Running a program in parallel on multiple processors—instead of serially on a single processor—can significantly reduce the amount of elapsed time required to run the program. Running in parallel, however, consumes more system resources (CPU time and memory) than running serially. Trading off a reduction in system throughput for a reduction in elapsed time is a decision that depends on the application being executed and the environment in which it is being executed.

To achieve maximum benefit, only *compute-intensive* code sequences should be considered for running in parallel. For example, program segments dealing with arithmetic operations performed on arrays (matrix arithmetic) are generally good candidates for parallel processing. You can identify other compute-intensive code segments using the VAX Performance and Coverage Analyzer (PCA) software product. After isolating code sequences that are candidates for parallel processing, you can then analyze the sequences in detail and make any coding changes that are necessary.

Mechanisms provided by VAX FORTRAN support parallel processing of *indexed DO loops*. Processing an indexed DO loop in parallel means that the iterations in the loop are divided among multiple processors and are executed concurrently.

NOTE

Throughout this chapter, indexed DO loops to be processed in parallel are referred to as *parallel DO loops*.

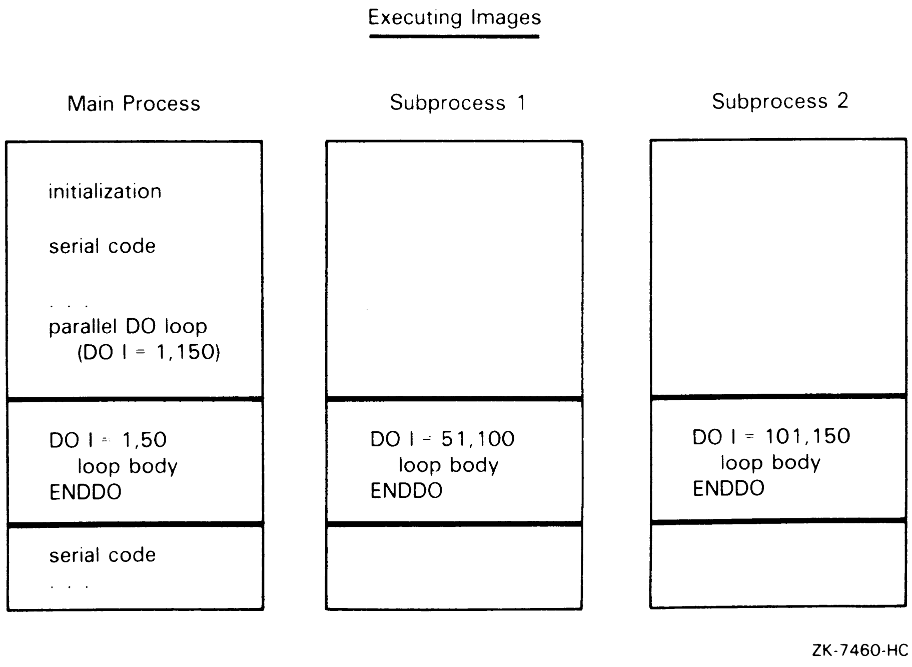
The compiler must decompose each parallel DO loop into groups of loop iterations that can be executed in parallel. Depending on the design and implementation of the compiler, this decomposition can either be handled automatically by the compiler or be specified manually by the programmer. VAX FORTRAN Version 5.0 supports *directed* decomposition, which is a variation of the manual method. With VAX FORTRAN, you direct the actions of the compiler using compiler directives, but the compiler hides the operating system and hardware mechanisms that are used to achieve parallelism. This means that you must determine where decomposition is safe (no unacceptable data dependences) and desirable (sufficient work in the loop). After performing this analysis, you must mark each individual DO loop that is to be decomposed and run in parallel. You must also make any coding changes that are needed to ensure correct results when iterations of a parallel DO loop are executed separately in parallel processes.

NOTE

> In many instances, you can prepare an indexed DO loop for parallel execution simply by identifying it as a parallel DO loop and allowing the parallel-processing defaults to take effect.

Unlike a normal, nonparallel, indexed DO loop, which is executed serially (that is, iterations are executed in sequence, lower bound through upper bound), a parallel DO loop is executed simultaneously in two or more processes, with each process executing a *segment* of the iterations in the loop. Figure 15–1 shows how a program containing an indexed DO loop might be executed in parallel. Note that the main process executes all of the nonparallel (serial) code in the program and that each process executes a 50-iteration segment of the iterations in the loop.

Figure 15–1: Processing an Indexed DO Loop in Parallel



The subprocesses are created during the initialization phase. They are not activated, however, until a parallel DO loop is encountered. When they complete the execution of their portion of the iterations in a parallel DO loop, they are placed in a wait state until the next parallel DO loop is encountered.

The order in which the individual iterations will complete is indeterminate; thus, no single iteration of the parallel DO loop can reliably depend on the values established in memory locations shared with other iterations. Understanding the implications this has for data use within a parallel DO loop is fundamental to an understanding of how to effectively use VAX FORTRAN parallel-processing support mechanisms. Section 15.3 provides a detailed analysis of the problems associated with the use of data in parallel DO loops.

Because of the overhead associated with parallel processing, a parallel DO loop should contain a total of 1000 or more instruction executions in order to produce a marked improvement in elapsed execution time. For example, a loop with two iterations and 500 instructions in each iteration meets this criteria. A loop with more iterations and fewer instructions in each iteration can also suffice, as in the following case:

```
DO I=1,100
    .
    .  [10 instructions]
    .
ENDDO
```

If a parallel DO loop invokes a subprogram containing another parallel DO loop, only the parallel DO loop of the calling program will be run in parallel. Each of the processes executing the outermost parallel DO loop will execute all of the iterations in the innermost parallel DO loop in a serial, nonparallel fashion.

VAX FORTRAN supplies the following mechanisms to assist parallel processing:

Table 15–1: VAX FORTRAN Parallel-Processing Support Mechanisms

FORTRAN Command Line Qualifier	
/[NO]PARALLEL	Use the /PARALLEL qualifier on your FORTRAN command line when compiling program units that are part of a program to be run in parallel. The use of the qualifier determines whether the compiler generates coding structures that are needed to support parallel execution.
Compiler Directive Statements	
DO_PARALLEL	Use the DO_PARALLEL directive to identify an indexed DO loop as a parallel DO loop and, optionally, to indicate how the loop iterations are to be divided up among the various processors.
SHARED CONTEXT_SHARED PRIVATE	Use the SHARED, CONTEXT_SHARED, and PRIVATE directives to control the sharing or non-sharing of memory locations during the execution of applications containing parallel DO loops.
LOCKON LOCKOFF	Use the LOCKON and LOCKOFF directives to ensure that certain statements within a parallel DO loop are executed in only one process at a time.
Run-time Environment Control Mechanisms	
FOR$PROCESSES FOR$SPIN_WAIT FOR$STALL_WAIT	Assign values to the logical names FOR$PROCESSES, FOR$SPIN_WAIT, or FOR$STALL_WAIT to adjust some aspects of the run-time environment in which your program will be executed.
Intrinsic Function	
NWORKERS	Use the NWORKERS intrinsic function, optionally, as an aid in dividing up iterations of a parallel DO loop among parallel processes.

Sections 15.7.1 to 15.7.4 describe in detail the mechanisms shown in this table.

Once you become familiar with VAX FORTRAN's parallel-processing support mechanisms, you will usually be able to quickly determine whether an indexed DO loop can be run in parallel. Then, in most instances, you will have to make only a few minor coding changes to prepare the loop for parallel processing.

15.2 Preparing Programs for Parallel Processing

Whether dealing with new or existing code, it is essential to fully understand how data is used (accessed or modified) within a parallel DO loop. It is also important to understand how the data is used in routines called from within a parallel DO loop and in code encountered after the completion of a parallel DO loop. Otherwise, it will not be possible for you to identify those indexed DO loops that can be executed in parallel without introducing erroneous or unpredictable results.

Developing a VAX FORTRAN parallel-processing program and preparing it for execution involves the following special steps and considerations:

1. Find the compute-intensive indexed DO loops in your program. Loops with a large number of iterations are always prime candidates. Those with a small number of iterations are candidates only if they involve extensive computational operations. (See Section 15.1.)
2. Determine whether the loops can be run in parallel by analyzing—in both parallel and nonparallel processing contexts—any variables that are defined in the loop. If necessary, recode or restructure the DO loops so that they can run in parallel and with optimum efficiency. (See Section 15.3.)
3. Ensure that variables and common blocks referenced within the parallel DO loop have the correct memory allocation attribute (shared or private) and that the attributes you establish for them are not in conflict with their attributes and use outside the parallel DO loop. By default, all common blocks are shared. Variables are also shared by default—unless they are referenced in a subprogram called from within a parallel DO loop. In the exception case, they are always treated as private. Note that loop control variables must be declared as private. (See Sections 15.3 and 15.7.2.2.)
4. Precede each parallel DO loop with a DO_PARALLEL directive, which identifies it as a parallel DO loop. (See Section 15.7.2.1.)
5. Compile the program with the /NOPARALLEL qualifier (default) and then run and debug the program, in serial mode, for any logic or coding errors.
6. After verifying that the program executes without errors in serial mode, recompile the program with /PARALLEL in effect. All routines from the main program through any routine in the call tree containing a parallel DO loop must be compiled with the /PARALLEL qualifier in effect. Other routines do not need to be compiled with the /PARALLEL qualifier unless they use shared common blocks.

Included in the routines that do not need to be compiled with the /PARALLEL qualifier are routines called (directly or indirectly) from within a parallel DO loop. These routines can be compiled with the /NOPARALLEL qualifier because they are executed serially within each of the processes executing the parallel DO loop. (See Section 15.7.1.)

7. Optionally, adjust the run-time environment by defining logical names that are used by the run-time environment during the execution of a program containing parallel DO loops. (See Section 15.7.3.)
8. Compare the results of serial execution of the program with the results of parallel execution. If necessary, debug the program while executing it in parallel mode. (See Section 15.5.)

The preceding list presents a general checklist of considerations that are involved in preparing VAX FORTRAN parallel-processing applications.

The sections that follow provide details on the following topics:

- Coding restrictions
- Coding options affecting execution efficiency
- Use of other languages within a VAX FORTRAN parallel-processing application
- Parallel processing effects on random number generators
- Parallel processing effects on exception handling

15.2.1 Coding Restrictions Within Parallel DO Loops

Several VAX FORTRAN language constructs and statements that are valid within nonparallel DO loops should not be used in parallel DO loops. Because iterations of a parallel DO loop are executed in an indeterminate order and in different processes, the use of any of these items within a parallel DO loop results in unpredictable run-time behavior.

15.2.1.1 Coding Restrictions Flagged by the Compiler

If the following coding restrictions for parallel DO loops are not observed, the compiler will generate a compile-time error. If these restrictions are not met in a source program, it will be impossible for the program to generate valid results.

- The loop control variable for a parallel DO loop must be of data type INTEGER.
- The loop control variable for the DO loop must be a private variable.
- The maximum length of a name for a shared common block is 26 characters.
- Only comment lines and blank lines can be placed between a DO_PARALLEL directive and the indexed DO loop statement to which it is coupled.
- I/O statements are not allowed within the body of a parallel DO loop.
- The control statements PAUSE and STOP are not allowed within the body of a parallel DO loop.
- Private symbols and common blocks referenced within the body of a parallel DO loop cannot be used in a SAVE statement.
- A DO loop with a branch (GOTO) into or out of its body cannot be run in parallel.

Additional coding restrictions associated with the use of compiler directives are described in Section 15.7.2.

15.2.1.2 Coding Practices that May Cause Unpredictable Results

Unpredictable results may occur if the following restrictions are not observed within code that executes in a parallel processing context, that is, within code that executes after entry into and before exit from a parallel DO loop.

- Resolving data-dependence problems is the major issue associated with parallel processing. Restrictions involving data dependence are as follows:
  - Data dependences involving shared variables must not exist between iterations of a parallel DO loop.
  - Data dependences involving private variables must not exist between code within a parallel DO loop and code executed before entry into or after the completion of the parallel DO loop.

See Section 15.3 for information on how to handle data dependences.

- System services or run-time library routines that change the context of a process (for example, a change in privileges, priority, access mode, or logical names) must not be called from within a parallel DO loop. (This includes calls to the run-time library's LIB$ESTABLISH routine.)
- The RETURN statement must not be used from within a parallel DO loop.
- I/O statements and the control statements PAUSE and STOP must not be used in a routine called—at any call level—from within a parallel DO loop.
- Private symbols and common blocks must not be referenced in a SAVE statement in a routine called—at any call level—from within a parallel DO loop.
- If a dummy argument is referenced within a parallel DO loop, the corresponding actual argument must reside in shared memory. This means that the caller of the routine containing the parallel DO loop (and any previous callers) must be compiled with the /PARALLEL qualifier.
- The use of random number generators within a parallel DO loop must be done with care because parallel processing affects how the numbers are generated. See Section 15.2.4 for a detailed description of how parallel processing affects random number generators.

The compiler does not flag any of these coding usages.

15.2.2 Coding Techniques for Improving Execution Efficiency

Coding techniques described in this section involve maximizing the work being done within a parallel DO loop and balancing the workload distributed among the processors executing the parallel DO loop.

Maximizing Workload Within Parallel DO Loops

As a general rule, the more work you include within the parallel DO loop, the more efficient the processing will be.

The following example illustrates a case in which an inner DO loop can be run in parallel and the outer DO loop cannot be run in parallel because of a data dependence problem. By interchanging the loops—and thus including more work within the parallel DO loop—processing is made more efficient.

Original DO Loop	Revised DO Loop
	`CPAR$ PRIVATE J,I`
	`      . . .`
	`CPAR$ CONTEXT_SHARED A`
	`CPAR$ DO_PARALLEL`
`DO I = 1,100`	`      DO J = 1,300`
`   DO J = 1,300`	`         DO I = 1,100`
`      A(I,J) = A(I+1,J) + 1`	`            A(I,J) = A(I+1,J) + 1`
`   ENDDO`	`         ENDDO`
`ENDDO`	`      ENDDO`

Balancing Workload Distribution Across Processors

By default, VAX FORTRAN distributes iterations to the processes in a way that works well when all of the iterations do about the same amount of work or when they contain unpredictable amounts of work. In special circumstances, such as the two described here, it may be advisable to manually adjust the distribution of loop iterations in order to balance the workload. To do this, you specify the size of the iteration segments to be distributed among the processes using the DO_PARALLEL directive.

The number of loop iterations executed by each of the parallel processes can, depending on the code within the loop, have a significant impact on execution efficiency. In some cases, the work being done inside the loop will influence how you should distribute it among the processes.

For example, in the following parallel DO loop, it might be very inefficient to divide the iterations into blocks of 50:

```
DO I = 1,100
...
IF (I .LT. 50) THEN
   CALL SUB(I)
ENDIF
...
ENDDO
```

In this situation, an iteration block size of 25 would probably be much more efficient on a system with two processors, depending on the amount of work being done by SUB.

In the next example, the amount of work done by the individual iterations is unbalanced (later iterations do less work). When patterns like this exist, manual assignment of the iterations can work better than taking the default.

```
      NPROCS = NWORKERS( )
CPAR$ DO_PARALLEL ( (N-K)+(2*NPROCS-1) ) / (2*NPROCS)
      DO J=K+1,N
        DO I=J,N
          A(I,J) = A(I,J) - A(J,K)*A(I,K)
        ENDDO
      ENDDO
```

The iterations are divided into 2*NPROCS segments. As a result, each process gets one iteration segment that contains a large amount of work and a second iteration segment that contains a small amount of work. In this way, you can balance the amount of work done in the parallel processes.

15.2.3 Use of Other Languages in Parallel-Processing Programs

The main program in an application that uses VAX FORTRAN parallel-processing support mechanisms must be a VAX FORTRAN program and must be compiled with the /PARALLEL qualifier.

Routines that do not invoke, directly or indirectly, a VAX FORTRAN routine containing a parallel DO loop can be written in any language and can be used without restrictions.

Those routines that do invoke, directly or indirectly, a VAX FORTRAN routine containing a parallel DO loop have the following restrictions:

- Actual arguments that correspond to dummy arguments referenced within a parallel DO loop must be accessible to all processes, that is, be declared as shared. Two methods of doing this with non-FORTRAN routines are as follows:
  - The routine declaring the actual argument allocates it on the stack.
  - The routine declaring the actual argument maps it to a shared common block.
- For routines not written in VAX FORTRAN and called from within a parallel DO loop, any static variables are treated as private by each process.

15.2.4 Use of Random Number Generators Within Parallel DO Loops

A random number generator is an example of a subprogram that keeps state across its calls, and thus is not normally safe for use within a parallel loop. The subprogram uses a seed to create a random number, and also produces a new seed value to be used on the next call. In VAX FORTRAN, the RAN function (described in the *VAX FORTRAN Language Reference Manual*) takes the seed as an argument, so the user can decide whether to use a shared variable or a local variable for the seed. This allows several choices for generating random numbers within a parallel loop. (Although this discussion concentrates on use of the RAN function, much of it applies to user-written functions and subroutines as well.)

A random number generator should produce sequences of values that are not correlated, over either a short span or a long span. Also, to simplify debugging, it is sometimes useful to be able to rerun a program with the same sequence of values.

The most conservative approach to using random numbers in a parallel loop is to precompute all of the random numbers the application will need by calling the RAN function in serial code before entering the parallel loop. Unfortunately, this is usually neither convenient nor efficient.

It is possible to call the RAN function within a parallel loop, but you must be careful. If the same variable is used as a seed in each iteration of a parallel loop, as in the following example, the results will not be satisfactory:

```
C -- Don't do this! --
CPAR$ CONTEXT_SHARED ISEED
CPAR$ DO_PARALLEL
      DO 20 J=1,100
        DO 20 I=1,100
20        A(I,J)=RAN(ISEED)
```

Nothing prevents several different processes from each reading the same value of the seed, so different columns of the resulting array will contain matching runs of values. (This example stores the random values so they could be examined; a real application would normally just do computations based on the random values. There is no need to store the values in an application program.)

One solution is to place LOCKON/LOCKOFF directives around uses of the RAN function. This has two disadvantages:

- The lock directives can significantly reduce the parallelism in the application if random numbers are heavily used.

- It is not possible to reproduce a run using the same val .es from the random number generator because the order in which different iterations seize the lock is unpredictable.

Another solution is to use a different seed for each iteration of the parallel loop. Calling RAN does not require synchronization if this is done, and the only problem is initializing the seed for each iteration. VAX FORTRAN's RAN function produces a sequence of random values that repeats after 2**32 calls; the initial seed selects a starting point within this sequence. If you use a different seed for each iteration of the parallel loop, you will want their initial values to be widely spaced across the sequence.

The coding in the following example produces poor spacing. It results in each iteration using the same sequence offset by one, so that for nearly every element A(I,J), there is another element A(I+1,K) with the same value:

```
C -- Don't do this! --
      COMMON /SEEDS/ ISEED(100)
CPAR$ SHARED /SEEDS/
C Initialize seeds -- poor way --
      ISEED(1)=1234567
      DO 10 I=2,100
        ISEED(I)=ISEED(I-1)
10      T=RAN(ISEED(I))
CPAR$ DO_PARALLEL
      DO 20 J=1,100
        DO 20 I=1,100
20        A(I,J)=RAN(ISEED(J))
```

The following technique gives well-spaced starting values for different seeds:

```
      SUBROUTINE INIT_SEEDS (SEEDS,N,ISEED)
      INTEGER N,ISEED,SEEDS(N)
C
C   This subroutine initializes an array of random-number generator
C   seeds so that they are about equally spaced along the 2**32 long
C   sequence of values that can be produced by VAX FORTRAN's random
C   number generator.
```

```
C
      REAL*16 RSPACE
      REAL*16 MAGIC,CUR_SEED
C
C   Parameters controlling VAX FORTRAN's random number generator.
      INTEGER MPLR,ADDEND
      PARAMETER (MPLR=69069, ADDEND=1)
c
C   Get the magic number that controls the spacing.  Avoid powers of 2.
      ISPACE = (2.0D0**32-1)/N
D     PRINT *,ISPACE
      MAGIC = RSPACE(ISPACE)
C
C   Fill the array of seeds.  To avoid overflows, treat integers that
C   are greater than 2**31-1 as negative.
      SEEDS(1) = ISEED
      CUR_SEED = ISEED
      IF (CUR_SEED.LT.0) CUR_SEED = CUR_SEED + 2.0Q0**32
      DO I=2,N
        CUR_SEED = ((MPLR-1)*CUR_SEED + ADDEND)*MAGIC + CUR_SEED
        CUR_SEED = MOD( CUR_SEED, 2.0Q0**32 )
        IF (CUR_SEED.GE.2.0Q0**31) THEN
          SEEDS(I) = CUR_SEED - 2.0Q0**32
        ELSE
          SEEDS(I) = CUR_SEED
        ENDIF
      ENDDO

      END


      FUNCTION RSPACE(SPACE)
      REAL*16 RSPACE
      INTEGER SPACE
C
C   This function computes a magic number used in initializing seeds
C   for the VAX FORTRAN random number generator.  The result of this
C   function can be used to determine the n'th random number.
C
C   We compute
C      ADDEND * MOD( (MPLR**SPACE - 1) / (MPLR-1), 2**32 )
C   by expanding ((MPLR-1)+1)**N using the binomial theorem.
C
C   Parameters controlling VAX FORTRAN's random number generator.
C   We depend on the fact that MPLR is odd (so MOD((MPLR-1)**16,2**32)=0)
      INTEGER MPLR,ADDEND
      PARAMETER (MPLR=69069, ADDEND=1)
```

```
C
C   We'll need exact products of 32-bit numbers, so use H_float temps
      REAL*16 POWN,SUM,BINOM,MOD_15
C
C   In order to compute binomial coefficients, we need a modulus that
C   is divisible by all the integers up to 15, as well as 2**32.
      MOD_15 = 2.0Q0**32
      DO I=3,15
        MOD_15 = MOD_15*I
      ENDDO
C
C   Compute ( ((MPLR-1)+1)**SPACE - 1) / (MPLR-1).  We expand using the
C   binomial theorem, but
C    - omit the first term (which is 1)
C    - drop terms after the 16th (since MOD((MPLR-1)**16,2**32)=0)
C    - divide the other terms by (MPLR-1)
C   We take the results modulo 2**32 often, so the numbers don't grow
C   too big.
C
      POWN  = 1.0Q0
      SUM   = 0.0Q0
      BINOM = 1.0Q0
      DO I=1,16
        BINOM = MOD( (BINOM * (SPACE-I+1))/I, MOD_15)
        SUM   = MOD(  SUM+BINOM*POWN,      2.0Q0**32 )
D       PRINT *,BINOM,POWN,SUM
D       PRINT *
        POWN  = MOD(  POWN*(MPLR-1),       2.0Q0**32 )
      ENDDO
C
C   Multiply by addend and return.
      RSPACE = MOD( SUM*ADDEND, 2.0Q0**32 )
      END
```

This technique has the following advantages:

- The random number generator can be called without interrupting the parallel loop.
- The sequence of values returned is reproducible.
- The same results are produced regardless of the number of processes used to execute the program.

15.2.5 Influence of Parallel Processing on Exception Handling

During the execution of a parallel DO loop, a condition handler established outside a parallel DO loop cannot take the action of unwinding the call stack beyond the routine containing the parallel DO loop.

Any unhandled exceptions that occur during the execution of a parallel DO loop have the effect of terminating all of the subprocesses.

15.3 Data Dependence Problems Caused by Parallel Processing

The major concern associated with running an indexed DO loop in parallel is *data dependence*. In general, a data dependence exists in a program whenever a particular memory location is accessed multiple times, with at least one access being a store operation (for example, an assignment).

The dependence is "carried" by an indexed DO loop if it is possible to access the same memory location on two or more iterations of that loop, with at least one access being a store operation.

If a data dependence is carried by an indexed DO loop, the results of running it in parallel will almost always differ from the results of running it serially.

To enable you to control some instances of data dependence problems associated with executing indexed DO loops in parallel, VAX FORTRAN supports the directives SHARED, CONTEXT_SHARED, PRIVATE, LOCKON, and LOCKOFF.

Within a parallel DO loop, the LOCKON and LOCKOFF directives isolate statements that must be run one at a time because of an unacceptable data dependence problem involving the use of a shared variable (scalar, array, or record) or a shared common block. See Section 15.7.2.3 for detailed descriptions of the LOCKON and LOCKOFF directives.

NOTE

The term *variable* is used in this chapter to denote data items that can be scalars, arrays, or records.

The SHARED, CONTEXT_SHARED, and PRIVATE directives govern how data items are shared (or not shared) among parallel processes. You use the PRIVATE directive to control the allocation of both common blocks and variables, the SHARED directive for common blocks only, and the CONTEXT_SHARED directive for variables only.

- The PRIVATE directive causes user-specified variables and common blocks to be private (nonshared). That is, each of the processes executing iterations of a parallel DO loop maintains unique copies of private variables and common blocks in separate memory locations that are not shared with the other processes.

Because private data items are not shared among the processes executing a parallel DO loop, they must always be defined in a loop iteration before they are used within that iteration. This is necessary because only the iterations of the parallel DO loop being executed by the main process would have access to the private data items defined before entry into the loop.

Similarly, private variables and common blocks should not be used after the completion of a parallel DO loop without being redefined. This is necessary because only the values of private data items established in the main process are accessible when serial processing resumes. Any values established by the subprocesses are lost.

- The SHARED directive causes user-specified common blocks to be shared among all of the processes executing a program—in both parallel and serial processing contexts.
- The CONTEXT_SHARED directive causes user-specified variables to be treated as shared or private variables, depending on the context in which they are used. Context-shared variables use the same memory location throughout any one invocation of a subprogram (subroutine or function), including any parallel DO loops contained within the subprogram. However, if the subprogram has several concurrent invocations (because it is invoked from within a parallel DO loop), each invocation will use different memory locations for its context-shared variables.

SHARED and CONTEXT_SHARED are the defaults for program units compiled with the /PARALLEL qualifier on the FORTRAN command line. PRIVATE and CONTEXT_SHARED are the defaults for program units compiled with the /NOPARALLEL qualifier (default) on the FORTRAN command line.

The memory allocation attribute of each common block—private or shared—must be the same within all compilation units in a program to be run in parallel. Thus, after analyzing how the common blocks in parallel DO loops are to be treated, you must then ensure that the usage is not in conflict with usage in other areas of the program.

Actual arguments that correspond to dummy arguments referenced within a parallel DO loop must be accessible to all processes, that is, be declared as shared.

See Section 15.7.2.2 for detailed descriptions of the SHARED, CONTEXT_SHARED, and PRIVATE directives.

The sections that follow describe how to treat various forms of data dependence problems:

- Section 15.3.1 provides information on acceptable forms of data dependence (which require use of the SHARED, CONTEXT_SHARED, or PRIVATE directives).
- Section 15.3.2 and 15.3.3 provide information on how to recode loops to avoid problems with unacceptable forms of data dependence and how to use locks (LOCKON and LOCKOFF directives).

15.3.1 Acceptable Forms of Dependence

Only three cases of data dependence are acceptable during the execution of a parallel DO loop. In these cases, the dependence is not carried across loop iterations. In all other cases of data dependence, unpredictable results will occur unless some form of synchronization is used to control access to the data. The acceptable cases are as follows:

- Temporary variables whose values are established inside the loop before being used in the same iteration and not used outside the loop.
- Read-only variables whose values are established before entry into the loop and not modified within the loop.
- Variables defined in only one loop iteration and not involved in a dependence that crosses loop iterations (that is, they are not used in any other iteration).

In all other cases of data dependence involving DO loops, the DO loops cannot be run in parallel with predictable results—unless one of the following actions is taken:

- You recode (transform) the loop in a way that allows it to be run in parallel with predictable results. This can be done either by converting an unacceptable data dependence into one of the acceptable data dependences described in the preceding list or by recoding the parallel DO loop and moving the unacceptable data dependence outside the loop.
- You use synchronization mechanisms—LOCKON and LOCKOFF directives—to control access to the memory location associated with the data dependence.

These techniques for resolving unacceptable cases of data dependence are described in Section 15.3.2 and Section 15.3.3.

Detailed explanations of the acceptable cases of data dependence are provided in the sections that follow.

15.3.1.1 Temporary Variables

A temporary variable is always defined before it is used in each iteration of a parallel DO loop. For example:

```
DO I = 1,100
   ...
   TVAR = A(I) + 2
   D(I) = TVAR + Y(I-1)
   ...
ENDDO
```

In this example, the memory location associated with TVAR is defined in each loop iteration before it is used. Thus, in any single loop iteration, TVAR always has a predictable value.

Directive Requirements

You use the PRIVATE directive to protect temporary variables that meet the following criteria:

- They are accessed within more than one DO loop iteration.
- They are not involved in a data dependence that crosses between any of the iterations. That is, they do not have any "loop carried" dependences.
- If they are used after the completion of the DO loop, they are redefined before they are used.

When you declare a variable as private, each process executing iterations of the DO loop uses a separate memory location in which to maintain the value of the variable. If you do not declare a variable as private, each process will share the same memory location. (Sharing a memory location might produce erroneous results because the order in which the various DO loop iterations access memory locations is indeterminate.)

Loop control variables are a prime example of variables that require a memory allocation attribute of private. See Section 15.7.2.2 for additional information about the PRIVATE directive.

15.3.1.2 Read-Only Variables

A read-only variable is defined before entry into a parallel DO loop and referenced—but not modified—within the loop. For example:

```
DO I = 1,100
   ...
   A(I) = B(I) + C(I)
   ...
ENDDO
```

In this example, the values in the memory locations associated with arrays B and C are never modified inside the loop.

Directive Requirements

You use the CONTEXT_SHARED directive to ensure that the values of read-only variables (established in the main process) are accessible to the subprocesses—unless the variable is in a common block. If the variable is in a common block, you use the SHARED directive to establish the common block, not the variable, as shared. (Note that the correct treatment of read-only variables can also be established by allowing the default behavior for the /PARALLEL qualifier to take effect.)

See Section 15.7.2.2 for additional information about the CONTEXT_SHARED and SHARED directives.

15.3.1.3 Variables Defined and Not Used

Using variables that are defined in only one iteration of an indexed DO loop and not used in any of the other iterations is commonly done in indexed DO loops performing array arithmetic. For example:

```
DO I = 1,100
   ...
   X(I) = X(I) + Y
   ...
ENDDO
```

In this example, none of the memory locations associated with X(I) is used in more than one loop iteration. In cases like this, data dependence never crosses from one loop iteration to another.

Directive Requirements

As with read-only variables, you must identify the variable in question as a context-shared variable by using the CONTEXT_SHARED directive or by allowing the default for the /PARALLEL qualifier to take effect. If the variable is not given a memory allocation attribute of shared, it cannot be accessed by code that executes after the completion of the loop. If the variable is in a common block, you must declare the common block as shared by using the SHARED directive or by taking the default. These declarations ensure that the values defined by each of the subprocesses are accessible to the main process after the completion of the loop. See Section 15.7.2.2 for additional information about the CONTEXT_SHARED directive.

15.3.2 Using Code Transformations to Resolve Dependences

Unless you use locks (LOCKON and LOCKOFF directives), an indexed DO loop cannot be run in parallel with predictable results whenever a loop-carried dependence occurs—that is, when a value is stored in a single memory location during one iteration and used during another. This restriction exists regardless of the order in which the use and store operations occur. This is because the decomposed form of the loop will not necessarily fetch from and store to all of the shared locations in the same sequence as the original form of the loop. In many cases, however, you can recode a DO loop in a way that eliminates this type of problem.

The sections that follow describe three coding techniques that can be used to eliminate data dependence problems that would otherwise require either the use of locks or the removal of an indexed DO loop from consideration as a candidate for parallel processing. The use of these techniques is preferable to the use of locks, especially for small loops, because the system overhead generated by locks can be large. The three coding techniques are loop alignment, code replication, and loop distribution.

All of the examples in the following sections are constructed to demonstrate the coding techniques, not to exemplify indexed DO loops that would be candidates for parallel processing.

15.3.2.1 Loop Alignment

Loop alignment entails converting references to a memory location so that the references within a loop iteration "align" with each other, eliminating memory location offsets that would cause data dependence to cross between loop iterations.

The code in the following DO loop demonstrates an alignment problem:

```
DO I = 2,N
   ...
   A(I) = B(I)
   C(I) = A(I+1)
   ...
ENDDO
```

The value in memory location A(I+1) is used in one loop iteration and then the next iteration stores another value into that location, referencing it as location A(I).

When the code is executed serially, the value in memory location A(I+1) is always used before another value is stored into that memory location. This is not true, however, when the code is executed in parallel. For example, if loop iterations 4 and 5 execute in separate processes and iteration 5 executes before iteration 4, the value that iteration 4 accesses from the memory location associated with A(I+1) will be the value established by iteration 5 in the memory location associated with A(I).

The way to remedy this dependence is to bring into "alignment" the two references to the memory location in array A, that is, the references to A(I) and A(I+1). This can be done by changing the second assignment statement as follows:

Original Statement	Revised Statement
`C(I) = A(I+1)`	`C(I-1) = A(I)`

This eliminates the data dependence problem associated with the previous references to memory locations in array A. However, to compensate for the change to the array reference, the loop control values may have to be adjusted and appropriate IF constructs may have to be added in order to achieve the same effect as the original loop.

It is also important, of course, to maintain the order in which memory locations are accessed. In this case, memory location A(I+1) in the original DO loop is used in one iteration and then stored into in the next iteration (as memory location A(I)). By aligning the references, each iteration operates on only one memory location and, in the original order of the operations, array A's memory locations are stored into before they are used. Thus, in the revised DO loop being prepared for parallel processing, the statement performing the use operation must be moved ahead of the statement performing the store operation in order to preserve the original order of these operations.

In the example given here, the following additional changes would have to be made to the loop:

Original DO Loop	**Revised DO Loop**
	`CPAR$ PRIVATE I`
	`...`
	`CPAR$ DO_PARALLEL`
`DO I = 2,N`	`      DO I = 2,N+1`
`   A(I) = B(I)`	`         IF (I .GT. 2) C(I-1) = A(I)`
`   C(I) = A(I+1)`	`         IF (I .LE. N) A(I) = B(I)`
`ENDDO`	`      ENDDO`

Alternatively, you could compensate for the change to the array reference by distributing certain statements outside the loop:

Original DO Loop	**Revised DO Loop**
	`CPAR$ PRIVATE I`
	`      IF (N .GE. 2) A(2) = B(2)`
	`CPAR$ DO_PARALLEL`
`DO I = 2,N`	`      DO I = 3,N`
`   A(I) = B(I)`	`         C(I-1) = A(I)`
`   C(I) = A(I+1)`	`         A(I) = B(I)`
`ENDDO`	`      ENDDO`
	`      IF (N .GE. 2) C(N) = A(N+1)`

Note that when statements are distributed outside the loop, tests must be made to control when those statements are to be executed. Otherwise, they would always be executed and that behavior would be in error.

Also, when using the loop alignment technique to resolve a data dependence, you should check to ensure that the coding changes that you make to bring one reference into alignment doesn't cause previously aligned references to become unaligned.

15.3.2.2 Code Replication

Code replication entails duplicating certain operations in order to eliminate a data dependence problem.

The following example illustrates a data dependence problem that can be resolved by code replication:

```
DO I = 2,100
   A(I) = B(I) + C(I)
   D(I) = A(I) + A(I-1)
ENDDO
```

This example contains a loop-carried dependence between memory locations A(I) and A(I-1). The value at memory location A(I-1) is not always predictable because, in some instances, it will not be defined in one loop iteration before another loop iteration attempts to use it. For example, if iterations 2-50 are executing in the main process and iterations 51-100 are executing in a subprocess, it can be assumed that loop iteration 51 will attempt to use memory location A(I-1) before loop iteration 50 has stored a value in that memory location, referencing it as memory location A(I).

To eliminate this problem, you can establish the value of A(I-1) in a new memory location and then eliminate the reference to the old memory location, substituting a reference to the duplicated memory location. For example, you could revise the DO loop as follows:

Original DO Loop	Revised DO Loop
	`CPAR$ PRIVATE I,TA`
	`      A(2) = B(2) + C(2)`
	`      D(2) = A(2) + A(1)`
	`CPAR$ DO_PARALLEL`
`DO I = 2,100`	`      DO I = 3,100`
`   A(I) = B(I) + C(I)`	`         A(I) = B(I) + C(I)`
`   D(I) = A(I) + A(I-1)`	`         TA = B(I-1) + C(I-1)`
`ENDDO`	`         D(I) = A(I) + TA`
	`      ENDDO`

In this situation, you simply compute the value of memory location A(I-1), store it into temporary variable TA, and replace the reference to A(I-1) with a reference to variable TA. (Note that variable TA must be declared as PRIVATE.)

Notice that some of the calculations are pulled out of the loop and the iteration count is modified. This is necessary because the reference to A(I) in the original loop used its original value, not one computed by B(I)+C(I). Use of the code replication technique will always require this type of modification in order to bring references back into alignment.

15.3.2.3 Loop Distribution

Loop distribution entails breaking down a loop with data dependence problems into several loops—one or more of which can be run in parallel. For example, consider the following DO loop:

```
DO I = 1,100
   A(I) = A(I-1) + B(I)
   C(I) = B(I) - A(I)
ENDDO
```

This loop can be distributed as follows:

```
DO I 1,100
   A(I) = A(I-1) + B(I)
ENDDO
DO I 1,100
   C(I) = B(I) - A(I)
ENDDO
```

Given these changes, the second loop can now be executed in parallel. The first loop, however, contains a linear recurrence and cannot be run in parallel without producing unpredictable results. This is because, in some instances during parallel execution, predictable values will not be defined in the memory locations associated with A(I-1) before the locations are accessed. For example, if loop iterations 1-50 are executing on one processor and loop iterations 51-100 are executing on another processor, it can be assumed in most cases that loop iteration 51 will attempt to access a value in memory location A(I-1) before iteration 50 has executed (and stored the necessary value at that location).

15.3.3 Using Locks to Resolve Dependences

In certain situations, you can use LOCKON and LOCKOFF directives within a parallel DO loop to resolve a data dependence problem involving common blocks or variables that reside in shared memory.

NOTE

You should use locks (LOCKON and LOCKOFF directives) only when you have no other recourse. Before you use locks, you should always try to eliminate a dependence problem by modifying your source code. Using locks negates some of the performance benefit achieved from parallel processing.

When you place these directives around a segment of code containing the data dependence, you ensure that the code segment is executed in only one process at a time—that is, you ensure that it cannot be executed in parallel. This can be useful in a variety of situations. For example, if an indexed DO loop is performing a summing operation, the statement updating the sum could be locked, as shown in the following code:

```
      LOGICAL*4 LCK
CPAR$ PRIVATE I,Y
CPAR$ CONTEXT_SHARED LCK,SUM
      LCK = .FALSE.
      .
      .
      .
CPAR$ DO_PARALLEL
      DO I = 1,1000
         Y = some_calculation
CPAR$ LOCKON LCK
         SUM = SUM + Y
CPAR$ LOCKOFF LCK
      ENDDO
```

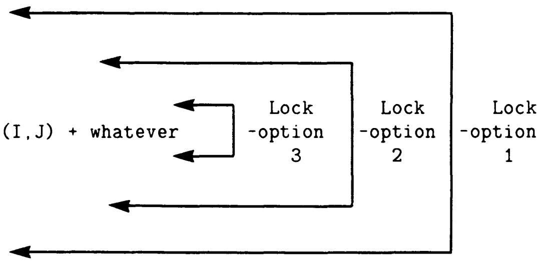
A lock is off if the lock variable has a value of .FALSE.; it is on if the lock variable has a value of .TRUE. In the previous example, LCK has the value .FALSE. when the LOCKON directive is first reached by one of the processes executing the parallel DO loop. The LOCKON directive determines that the lock is off (.FALSE.), sets the lock on (.TRUE.), and allows execution to continue into the code following the LOCKON directives. When the next process reaches the LOCKON directive, it must wait until the lock is unlocked. This is done when the earlier process executes the LOCKOFF directive, and the waiting process is then able to pass through the lock. The operation of the lock in this case is essentially a toggling operation in which the lock is alternately turned on by the LOCKON directive and off by the LOCKOFF directive as these directives are executed by the various processes executing iterations of the parallel DO loop.

In this instance, the lock ensures approximately the same results as a serial execution of the loop. The results may differ slightly because differences in the orders in which the various values of Y are added to SUM may cause the result to be rounded off differently. In parallel execution, this order is unpredictable, and generally different from the order of a serial execution of the loop.

The benefit from using locks is that the code preceding the lock is able to execute in parallel—while the locks provide serial control over a data dependence that would otherwise produce unacceptable results.

The following example illustrates a situation in which locks could be set at several points within a parallel DO loop.

```
CPAR$ DO_PARALLEL
      DO L = 1, N
      .
      .   [work]
      .
CPAR$ LOCKON LK1
         DO J = 1, 100
CPAR$ LOCKON LK2
            DO I = 1, 100
CPAR$ LOCKON LK3
               A(I,J) = A(I,J) + whatever
CPAR$ LOCKOFF LK3
            ENDDO
CPAR$ LOCKOFF LK2
         ENDDO
CPAR$ LOCKOFF LK1
      ENDDO
```



Options shown in this example are as follows:

- Lock the entire matrix—Option 1.

- Lock one column—Option 2.
- Lock one element—Option 3.

The point at which the lock is set should optimally result in the least amount of lock overhead and the greatest amount of parallel processing. In this instance, option 1 has the lowest overhead and option 3 has the highest parallelism. Option 2, however, provides the best balance between high parallelism and low overhead—with one hundred less overhead than option 3 and one hundred times more parallelism than option 1.

See Section 15.7.2.3 for details on the coding requirements associated with LOCKON and LOCKOFF directives.

15.4 Tuning Issues Related to Parallel Processing

Parallel-processing programs may fail to execute because of insufficient system resources. You may have to adjust some resource-utilization parameters—both for the entire system and for individual user accounts. You may also want to adjust some parameters in order to achieve better performance for programs executing in parallel. These types of considerations are addressed in the sections that follow.

You may also find it advisable to adjust system resources to accommodate the needs of the multiprocessing configuration of the VMS Debugger. System management considerations related to the debugger are described in Section A.3.

15.4.1 System Parameters Set with the SYSGEN Utility

When a parallel application is executed, much of the local memory and many common blocks of the application are mapped to global sections (VMS's way of sharing data between processes). Users must ensure that the number of global sections, global pages, and global page file sections required by a parallel application are available. To allow enough space for this global data, some of the system's sysgen parameters may need to be increased. The three sysgen parameters that are most important are GBLPAGFIL, GBLPAGES, and GBLSECTIONS. Note that these parameters are not dynamic; you must reboot your system in order for any modifications to them to take effect. You should adjust the parameters one at a time in order to avoid modifying some of them unnecessarily.

You should use the SYS$UPDATE:AUTOGEN.COM command procedure to modify these parameters. Using AUTOGEN.COM, parameters related to those you are modifying are changed for you automatically. For details on how to use this procedure, refer to the installation guide for the operating system software installed on your system.

Table 15-2: Sysgen Parameters Requiring Changes for Parallel Processing

Parameter Name	Current[1]	Default	Minimum	Maximum	Unit	Dynamic
GBLSECTIONS	512	128	20	4095	Sections	No
GBLPAGES	32768	4096	512	-1	Pages	No
GBLPAGFIL	7000	1024	128	-1	Pages	No

[1]Values listed under this heading are typical values.

GBLSECTIONS—Global section descriptor count

If the count is not high enough, the following diagnostic message is issued:

```
%SYSTEM-F-GSDFULL, global section descriptor table is full
```

The GBLSECTIONS parameter sets the number of global section descriptors established in permanently resident memory at bootstrap time. Each global section must have a descriptor. Thus, the number of global section descriptors determines the maximum number of global sections that can exist on the system at any one time.

Each descriptor requires 32 bytes of permanently resident memory. To avoid wasting permanently resident memory, you should attempt to minimize the value you give to the GBLSECTIONS parameter.

GBLPAGES—Global page table entry count

If the count is not high enough, the following diagnostic message is issued:

```
%SYSTEM-F-GPTFULL, global page table is full
```

The GBLPAGES parameter establishes the size of the global page table and the maximum number of global pages that can be created. For every 128 entries in the global page table, four bytes are added to permanently resident memory in the form of a system page table entry. (When you increase GBLPAGES beyond the default setting, you may want to increase the SYSMWCNT by one for each multiple of 128 entries that you add to the default setting.)

One way of calculating the number of global pages required to run an application using the VAX FORTRAN parallel processing support is to obtain a LINK map and add up the size of the PSECTs that will be shared.

To get a link map, specify the /MAP/FULL qualifiers on your LINK command line. To calculate the approximate number of global pages required for your application, go through the link map and add up the decimal sizes of the PSECTs for shared COMMON blocks and the $LOCAL PSECT. (The link map gives you the size of PSECTs in bytes.) In addition, the VAX FORTRAN parallel processing run-time support requires approximately 3 global pages for its own use, so add 1536 bytes to the number of bytes required for the PSECTs. Then, to determine the number of global pages required for the application, divide the total number of bytes by 512.

The GBLPAGFIL and GBLPAGES parameters must both be at least as large as the number of global pages required for your application.

You can minimize the amount of global memory required by an application by specifically declaring any common blocks as private if it is not necessary for them to be shared. The default memory attribute for common blocks is shared in program units compiled with the /PARALLEL qualifier, and, as noted previously, shared common blocks are mapped to global sections.

GBLPAGFIL—Global page file limit

If the limit is not high enough, the following diagnostic message is issued:

```
%SYSTEM-F-EXGBLPAGFIL, exceeded global page file limit
```

The GBLPAGFIL parameter establishes the maximum number of global pages with page file backing store that can be created. Global page file sections are allocated from the paging file at bootstrap time. When you increase this parameter you may want to increase the size of the paging files as well. The current size of the paging files can be seen using the DCL command SHOW MEMORY. For example:

```
$ SHOW MEMORY

 [ other memory information removed ]

Paging File Usage (pages):                 Free  Reservable      Total
  DISK$PAGE:[PAGE]SWAPFILE2.SYS;1          68280       68280      79992
  DISK$PAGE:[PAGE]PAGEFILE2.SYS;1          73490       60190      79992
```

15.4.2 User Parameters Set with the Authorize Utility

Two of the authorization quotas, the PRCLM and PGFLQUO quotas, may need to be adjusted for any account that will be running parallel applications.

- The PRCLM quota determines the number of subprocesses that a user's process can create. For applications involving parallel DO loops, it must be at least equal to the number you specify for the FOR$PROCESSES logical name. (During debugging operations, one additional process must be available for the debugger.)
- The PGFLQUO quota is a pooled quota. It restricts the total pages that user processes can use in the system paging file. It is shared by all processes in a job and thus may require an adjustment to allow for the additional processes used in parallel processing. It may need to be as high as the value that results from multiplying the total number of writable pages (shown in the Image Section Synopsis in the image map produced by the linker) times the number of processes.

If either of these quotas is not high enough, the following diagnostic message is issued:

```
%SYSTEM-F-EXQUOTA, exceeded quota
```

These quotas are adjusted using the VMS Authorize Utility and are established only at login time. This implies that any current user of the account must log off and back on again before the quotas will change for that user. The following user listing shows example settings for the PRCLM and PGFLQUO quotas:

```
Username: USER_J                                   Owner:  Joe User
Account:  NONE                                     UIC:    [360,100] ([USER_J])
CLI:      DCL                                      Tables: DCLTABLES
Default:  USRD$:[USER_J]
.
.  [ other user information removed from this listing display ]
.
Prclm:          10  DIOlm:          18  WSdef:          300
Prio:            4  ASTlm:          30  WSquo:          500
Queprio:         0  TQElm:          20  WSextent:      2048
CPU:        (none)  Enqlm:         200  Pgflquo:      20000
.
.  [ other user information removed from this listing display ]
.
```

Use the Authorize Utility's MODIFY command to change these quotas. For example:

```
UAF> MODIFY USER_J/PGFLQUOTA=20000
```

15.4.3 Other Tuning Considerations

Parallel processing applications typically use large amounts of memory. To get better performance for an application, you may find it advisable to make adjustments to the working set size parameters (WSMAX, WSQUOTA, WSEXTENT)—both for the system and for user accounts. Refer to the *Guide to VMS Performance Management* for information on how to adjust working set size.

15.5 Debugging Programs with Parallel DO Loops

This section provides a brief introduction into how to debug a VAX FORTRAN parallel-processing application—the steps you must take and the considerations involved.

Section 3.5.13 lists the debugger commands used to control a multiprocessing debugging session. Details about the use of the commands are available in the online help provided for the debugger. An overview of the features of the multiprocess debugging configuration is provided in Appendix A. (Note: The overview provided in Appendix A is not oriented to VAX FORTRAN applications, but to parallel processing in general.)

To test a new or revised program containing parallel DO loops, you should first compile it with the /NOPARALLEL qualifier, and run and debug it serially for any logic and coding errors. (Note: You can serially debug a program compiled with the /PARALLEL qualifier if you have set FOR$PROCESSES=1 (see Section 15.7.3). In this way, you can alternate between serial and parallel debugging—without recompiling.)

You should also specify bounds checking (/CHECK=BOUNDS) when compiling your program. This can be helpful because array references that are outside the boundaries of an array declaration may produce results that differ in serial and parallel executions of the code in which they occur. To resolve problems with array references that are out of bounds, you should either eliminate the out of bounds condition or analyze the dependences associated with the references and transform your source code as necessary.

After eliminating logic and coding errors detected during the serial execution of your program, you should then recompile the program with the /PARALLEL qualifier and run the program in parallel. If errors occur (that is, if results differ between serial and parallel executions), examine the use of variables and common blocks in the source code, looking for problems with data dependences that are carried across parallel DO loop iterations or into or out of a parallel DO loop. (See Section 15.3 for information on data dependence problems.) If you are able to isolate one or more data dependence problems, change the source code as necessary and then recompile the program again and repeat the parallel test run.

If you are unable to fix the errors that occur during the parallel execution of your program by analyzing and changing your source code, you will have to use the debugger to isolate the errors while your program is executing in parallel.

Example 15-1 illustrates a program that requires debugging. The program was compiled and linked without diagnostic messages from either the compiler or the linker. It executes properly when run serially. However, the program produces erroneous results when it is run in parallel because the array Z and the variable TEMP are declared improperly. (Z must be shared because different locations in it will be modified in parallel iterations of the loop, and TEMP must be private because it is modified and used in parallel iterations of the loop.)

Example 15–1: Sample VAX FORTRAN Parallel-Processing Source Program

```
C       A matrix multiplication program
C
C       This program demonstrates that
C          A * I = A (where I is an identity matrix)
C
      PROGRAM MATRIX_MUL

      INTEGER M,N,I,J,K,TEMP
      PARAMETER( M= 3, N= 3 )        ! use small number for debugging

      COMMON /MATRIX/ X,Y
      COMMON /MATZ/ Z

      INTEGER X(M,N),Y(N,N),Z(M,N)

CPAR$ SHARED /MATRIX/
CPAR$ PRIVATE /MATZ/,I,J,K

C         Initialize the X and Z matrices
C         and the Y matrix to be the identity matrix

      DO 100 J=1,N
          DO 100 I=1,M
              Z(I,J) = 0
100           X(I,J) = I * (J + 10)

      DO 130 J=1,N
          DO 130 I=1,N
130           Y(I,J) = 0

      DO 150 J=1,N
150       Y(J,J) = 1

CPAR$ DO_PARALLEL 1
      DO 200 J = 1, N
          DO 250 I = 1, M
              TEMP = 0
              DO 300 K = 1, N
                  TEMP = TEMP + X(I,K)*Y(K,J)
300               CONTINUE
250           Z(I,J) = TEMP
200   CONTINUE

      WRITE(*,*) X
      WRITE(*,*) Z

      END
```

The key to parallel debugging is to compare what happens to variables in the parallel DO loop iterations in which they are either used or modified.

Example 15–2 illustrates a typical terminal dialog for a debugging session involving the program shown in Example 15–1. This example shows how to control execution in two parallel processes and how to switch back and forth between the processes. Underlining is used in the example to indicate user input. The highlighted numbers in the example dialog are keyed to notes that explain the debugging operations being performed.

Example 15–2: Sample Parallel-Processing Debugging Session

```
$ FORTRAN/DEBUG/NOOPT/PARALLEL MATRIX
$ LINK/DEBUG MATRIX
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS                              ❶
$ RUN MATRIX

      VAX DEBUG VERSION T5.0-00 MP

%DEBUG-I-INITIAL, language is FORTRAN, module set to MAIN                    ❷
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 1
DBG_1> go
!predefined trace on activation at FOR$INIT_PARALLEL+36 in %PROCESS_NUMBER 2
!break at routine MATRIX_MUL in %PROCESS_NUMBER 1
!    22:         DO 100 J=1,N
DBG_1> show process/all                                            ❸
! Number  Name          Hold  State           Current PC
!*    1 HUANG                 break           MATRIX_MUL\%LINE 22
!     2 FOR$20200DD5_01       interrupted     475018
```

Example 15–2 Cont'd. on next page

Example 15–2 (Cont.): Sample Parallel-Processing Debugging Session

```
DBG_1> do (set break %line 36)                                    ❹
DBG_1> go
!break at MATRIX_MUL\%LINE 36 in %PROCESS_NUMBER 2
!    36:              DO 250 I = 1, M
!break at MATRIX_MUL\%LINE 36 in %PROCESS_NUMBER 1
!    36:              DO 250 I = 1, M
DBG_1> set process/hold/all                                       ❺
DBG_1> do (set watch temp; set break %line 41)                     ❻
DBG_1> go
!watch of MATRIX_MUL\TEMP at MATRIX_MUL\%LINE 39+4 in %PROCESS_NUMBER 1
!    39:                      TEMP = TEMP + X(I,K)*Y(K,J)
!   old value: 0
!   new value: 11
!break at MATRIX_MUL\%LINE 40 in %PROCESS_NUMBER 1
!    40: 300              CONTINUE
DBG_1> go
!break at MATRIX_MUL\%LINE 41 in %PROCESS_NUMBER 1
!    41: 250              Z(I,J) = TEMP
DBG_1> examine temp
!MATRIX_MUL\TEMP:         11
DBG_1> set process %proc 2                                        ❼
DBG_2> go
!watch of MATRIX_MUL\TEMP at MATRIX_MUL\%LINE 37 in %PROCESS_NUMBER 2
!    37:                  TEMP = 0
!   old value: 11
!   new value: 0                                                              ❽
!break at MATRIX_MUL\%LINE 38 in %PROCESS_NUMBER 2
!    38:                  DO 300 K = 1, N
DBG_2> go
!watch of MATRIX_MUL\TEMP at MATRIX_MUL\%LINE 39+4 in %PROCESS_NUMBER 2
!    39:                      TEMP = TEMP + X(I,K)*Y(K,J)
!   old value: 0
!   new value: 12
!break at MATRIX_MUL\%LINE 40 in %PROCESS_NUMBER 2
!    40: 300              CONTINUE
DBG_2> go
!break at MATRIX_MUL\%LINE 41 in %PROCESS_NUMBER 2
!    41: 250              Z(I,J) = TEMP
```

Example 15–2 Cont'd. on next page

Example 15–2 (Cont.): Sample Parallel-Processing Debugging Session

```
DBG_2> examine temp
!MATRIX_MUL\TEMP:        12
DBG_2> step
!stepped to MATRIX_MUL\%LINE 37 in %PROCESS_NUMBER 2
!    37:                 TEMP = 0
DBG_2> examine z
!MATRIX_MUL\Z
!    (1,1):      0
!    (2,1):      0
!    (3,1):      0
!    (1,2):      12
!    (2,2):      0
!    (3,2):      0
!    (1,3):      0
!    (2,3):      0
!    (3,3):      0
DBG_2> set process %proc 1                                   ❾
DBG_1> step
!stepped to MATRIX_MUL\%LINE 37 in %PROCESS_NUMBER 1
!    37:                 TEMP = 0
DBG_1> examine z
!MATRIX_MUL\Z
!    (1,1):      12
!    (2,1):      0
!    (3,1):      0
!    (1,2):      0
!    (2,2):      0
!    (3,2):      0
!    (1,3):      0
!    (2,3):      0
!    (3,3):      0
DBG_1> quit
```

Notes to Example 15–2:

❶ Establish a multiprocessing debugging configuration before invoking the debugger.

❷ The messages issued at the start of a multiprocess debugging session differ from those issued for a single-process debugging session because of special initialization done for the multiprocessing configuration. Also, a predefined tracepoint is triggered whenever a new process comes under debugger control.

❸ Entering the SHOW PROCESS command shows that a subprocess has been created and is now under debugger control.

❹ Set a breakpoint on the first line inside the parallel DO loop that appears to be causing the problem. (The DO command broadcasts the debugger command specified in parentheses to all of the processes.)

❺ Setting all of the processes in a hold state confines execution to the visible process. This allows you to control execution of each process separately.

❻ Set a watchpoint on a variable that may be causing the problem. Also, set a breakpoint on the place where its value is used. (Note: When execution is suspended in any process, execution in all other processes is interrupted as well. By default (SET PROCESS/DYNAMIC), the process in which execution is suspended is automatically established as the visible process.)

❼ Watch execution in process 1. Notice that TEMP has a value of 11 before it is stored in matrix Z.

❽ Switch to the other process to watch the execution of the same code. Notice that the value of TEMP that process 1 is going to use has been overwritten. Also notice that the prompt has changed—from DBG_1 to DBG_2. A command is executed only in the context of the visible process, unless it is broadcast to other processes by means of the DO command.

❾ Now, switch back to process 1 and compare the contents of array Z. Notice that the value placed in the array by process 2 is not in the process 1 display. This indicates that you have a sharing problem. If Z is not shared, the results of computations in process 2 will be lost.

Also, note that process 1 has stored the value 12 instead of the value 11 in Z(1,1). This indicates another sharing problem: process 2 has overwritten the value established by process 1 in the variable TEMP, and thus process 1 stores the wrong value in array Z. The variable TEMP must be declared as a private variable.

At this point, the correct contents of matrix Z should be as follows:

```
(1,1):     11
(2,1):     0
(3,1):     0
(1,2):     12
(2,2):     0
(3,2):     0
(1,3):     0
(2,3):     0
(3,3):     0
```

15.6 Sample Use of Parallel Processing

This section contains an example of the use of parallel DO loops in an application involving matrix multiplication and several advanced examples of conversions of DO loops involving linear recurrences.

15.6.1 Matrix Arithmetic

DO loops containing array operations are prime candidates for parallel processing. The following program shows the use of parallel DO loops in an application involving matrix multiplication:

```
      PROGRAM MATMUL
      INTEGER M,N
      PARAMETER( M= 200, N= 250 )

C The matrix arrays must be shared
C
CPAR$ SHARED /MATX/
      COMMON  /MATX/  X,Y,Z
      REAL  X(M,N),Y(N,N),Z(M,N)

C The control variables for the parallel DO loop and the DO loops
C nested in the parallel DO loop must be PRIVATE
C
CPAR$ PRIVATE I,J,K
      INTEGER*4 I,J,K

      REAL       MTH$RANDOM
      INTEGER*4  SEED
C
C Initialize the X matrix first

      SEED = SECNDS( 0. )

      DO 100 J=1,N
        DO 100 I=1,M
100     X(I,J) = MTH$RANDOM( SEED )

C Now initialize the Y matrix to the identity matrix

      DO 105 J=1,N
        DO 105 I=1,N
105     Y(I,J) = 0

      DO 110 J=1,N
110   Y(J,J) = 1
```

```
C Do the outer most DO loop in parallel
C
CPAR$ DO_PARALLEL
C
      DO 220 J = 1,N
        DO 210 I=1,M
210       Z(I,J) = 0.0
        DO 220 K=1,N
          DO 220 I=1,M
            Z(I,J) = Z(I,J) + X(I,K)*Y(K,J)
220   CONTINUE

C Verify the results
C
      DO 155 J=1,N
        DO 155 I=1,M
          IF ( X(I,J) .NE. Z(I,J) ) THEN
160         WRITE(*,*) 'Error in matrix multiplication...'
            CALL EXIT( 1 )
          ENDIF
155   CONTINUE

      CALL EXIT( 1 )

      END
```

15.6.2 Linear Recurrences

Some dependences in a DO loop involve linear recurrences, that is, situations in which the value derived from a statement depends on an earlier execution of that statement. For example:

```
C(I) = C(I-1) + B
```

Some DO loops with recurrences can be recoded for parallel execution. However, this is more difficult (and often less productive) than the other techniques described in Section 15.3.2.

The examples shown here involve loops with linear recurrences. Transforming these loops requires a significant effort because the order in which the arithmetic operations are performed in the loops can never be guaranteed to be the same for both serial and parallel execution. However, even though the arithmetic operations are performed in different order, you can achieve the same results in some cases by transforming the loops. (Final results may differ slightly because of roundoff differences introduced by varying the order of the arithmetic operations.)

NOTE

A loop with linear recurrences should be considered as a candidate for parallel processing only under the following circumstances:

- You determine that it is extremely compute intensive and will thus benefit markedly from parallel processing.
- It is not contained within an outer loop that can be more easily prepared for parallel processing.

Otherwise, it should not be considered for transformation because resolving a recurrence problem can be very difficult.

The original DO loops in the examples are part of the "Livermore Loops" program prepared at Lawrence Livermore National Laboratory, Livermore, California.[1]

The examples are Kernel 3 (Inner Product) and Kernel 5 (Tri-Diagonal Elimination, Below Diagonal). Other kernels involving linear recurrences can be converted with the same techniques shown here. Note that many of the kernels in this program do not involve recurrences and can be converted without difficulty.

The following variables and directives would have to be added to the subroutine containing the kernels in order to support the DO loops in their parallel forms:

```
C
C     Extra variables and directives required for parallel execution
C
      REAL*8  SP,TP
      REAL*8 SUMS(0:31),PRODS(0:31)
      INTEGER LCHUNK,ICHUNK,NPROCS
CPAR$ SHARED_ALL
CPAR$ PRIVATE I,J
CPAR$ PRIVATE SP,TP
      NPROCS = NWORKERS ( )         !Establish the number of
                                    !available processors
```

[1] The "Livermore Loops" program is copyrighted by and reproduced with the permission of the Regents of the University of California.

Kernel 3

The DO loop in Kernel 3 (Inner Product) in its serial form appears as follows:

```
       ...
       Q= 0.0
       DO 3 K= 1,N
3        Q= Q + Z(K)*X(K)
       ...
```

In its parallel form, the loop could be transformed as follows:

```
        ...
C     Parallel part, phase 1.  Compute one partial sum in each worker.
C
      LCHUNK = (N+NPROCS-1)/NPROCS
CPAR$ DO_PARALLEL 1
      DO I=0,NPROCS-1
        TP=0.0
        DO J=I*LCHUNK+1, MIN(N,I*LCHUNK+LCHUNK)
          TP=TP+Z(J)*X(J)
        ENDDO
        SUMS(I)=TP
      ENDDO

C     Serial part, phase 2.  Add up the partial sums.
C
      Q=SUMS(0)
      DO I=1,NPROCS-1
        Q=Q+SUMS(I)
      ENDDO
```

Kernel 5

The DO loop in Kernel 5 (Tri-Diagonal Elimination, Below Diagonal) in its serial form appears as follows:

```
      ...
      DO 5 I = 2,N
5       X(I)= Z(I)*(Y(I) - X(I-1))
      ...
```

In its parallel form, the loop could be transformed as follows:

```
      LCHUNK = 2*N / (2*NPROCS+3)
      ICHUNK = N - NPROCS * LCHUNK
```

```
C     Parallel part, phase 1
C
CPAR$ DO_PARALLEL 1
      DO J = 0,NPROCS-1
        IF (J .EQ. 0) THEN
           TP = X(1)        ! Completely solve first chunk.
           DO I = 1, ICHUNK-1
             TP = Z(1+I)*( Y(1+I)-TP )
             X(1+I) = TP
           END DO
        ELSE         ! Derive coefficients form remaining chunks.
           SP = 1.0
           TP = 0.0
           DO I = ICHUNK+J*LCHUNK-LCHUNK, ICHUNK+J*LCHUNK-1
             SP = -Z(1+I)*SP
             TP = Z(1+I)*( Y(1+I)-TP )
           END DO
           PRODS(J) = SP
        END IF
        SUMS(J) = TP
      END DO

C     Serial part, phase 2. Combine coefficients to get starting points.
C
      DO J = 1,NPROCS-1
        SUMS(J) = PRODS(J)*SUMS(J-1) + SUMS(J)
      ENDDO

C     Parallel part, phase 3. Solve each remaining chunk based on
C     its starting point.
C
CPAR$ DO_PARALLEL 1
      DO J = 1,NPROCS
        TP = SUMS(J-1)
        DO I = ICHUNK+J*LCHUNK-LCHUNK, ICHUNK+J*LCHUNK-1
          TP = Z(1+I)*( Y(1+I)-TP )
          X(1+I) = TP
        END DO
      END DO
       ...
```

15.7 VAX FORTRAN Parallel-Processing Support Mechanisms

VAX FORTRAN provides the following mechanisms for use with parallel-processing applications:

- A qualifier on the FORTRAN command line:

  /PARALLEL

- Compiler directive statements:

  DO_PARALLEL
  PRIVATE
  SHARED
  CONTEXT_SHARED
  LOCKON
  LOCKOFF

- Logical names:

  FOR$PROCESSES
  FOR$SPIN_WAIT
  FOR$STALL_WAIT

- An intrinsic function:

  NWORKERS

These parallel-processing support mechanisms are described in detail in the sections that follow.

15.7.1 /PARALLEL Qualifier on FORTRAN Command Line

The /PARALLEL qualifier directs the VAX FORTRAN compiler to perform special processing required for program units in a program to be run in parallel.

The /PARALLEL qualifier does not have any arguments. It has the form:

```
FORTRAN /[NO]PARALLEL [/other-qualifiers] filename.FOR
```

Specifying /NOPARALLEL has the same effect as omitting the /PARALLEL qualifier; that is, compiler directive statements relating to parallel processing (DO_PARALLEL, PRIVATE, SHARED, CONTEXT_SHARED, LOCKON, LOCKOFF) are treated as comments and no object code to support execution of parallel DO loops is generated.

Specifying /PARALLEL causes the compiler to interpret parallel-processing compiler directive statements in the source code, to generate object code to support execution of parallel DO loops, and to give memory allocation attributes of shared and context-shared to common blocks and variables, respectively.

Compilation units containing subprograms (subroutines or functions) that meet the following criteria must be compiled with the /PARALLEL qualifier:

- All subprograms that call, directly or indirectly, a subprogram containing a parallel DO loop
- All subprograms that contain parallel DO loops
- All subprograms that reference data items within common blocks established as shared within other subprograms

When a compilation unit is compiled with a /PARALLEL qualifier, all variables and common blocks are treated, by default, as follows:

- All common blocks declared in the compilation unit are mapped to a global section. This allows them to be shared among all of the processes running in parallel.
- All variables declared in the subprograms are context-shared. They are handled in a way that allows them to be either shared among the parallel processes or private to each parallel process, depending on where the subprogram appears in the call tree, that is, its call level. All variables in or above the subprogram call level containing the parallel DO loop are shared by default; all variables in subprograms called directly or indirectly from within the parallel DO loop are private. (All subprograms called directly or indirectly from within parallel DO loops execute serially within the process executing the loop iteration from which the call originates.)

Use the PRIVATE directive to override the default statuses that are given to variables (context-shared) and common blocks (shared) by the /PARALLEL qualifier.

15.7.2 Compiler Directives for Parallel Processing

To support parallel processing, VAX FORTRAN provides the following compiler directive statements:

DO_PARALLEL
SHARED[_ALL]
CONTEXT_SHARED[_ALL]
PRIVATE[_ALL]
LOCKON
LOCKOFF

As described in the chapter on compiler directives in the *VAX FORTRAN Language Reference Manual,* directives are prefixed, starting in column 1, with a 5-character identifier and a space (or tab). In the case of parallel directives, the 5-character identifier is CPAR$. For example:

```
CPAR$ PRIVATE_ALL
```

Unless the /PARALLEL qualifier is specified on the FORTRAN command line for the program unit, all of these directives are interpreted as comment lines.

A compiler directive cannot be continued across multiple lines in a source program, and any blanks appearing after column 6 are insignificant.

15.7.2.1 DO_PARALLEL Directive

The DO_PARALLEL directive identifies an indexed DO loop that is to be executed in parallel.

The directive has the format:

```
column 1
|
CPAR$ DO_PARALLEL [distribution-size]
```

The DO_PARALLEL directive must precede the DO statement for each parallel DO loop. No source code lines, other than comment lines and blank lines, can be placed between the DO_PARALLEL directive and the DO statement.

You can specify how the DO loop iterations are to be divided up among the processors executing the parallel DO loop. For example, if a parallel DO loop has 100 iterations and you specify a distribution size of 25, iterations will be distributed to each processor for execution in sets of 25. When a process completes one set of iterations, it then begins processing the next set. If the number that you specify for distribution size does not divide evenly into the number of iterations, any remaining iterations are run in the last process.

The expression that you use to specify the distribution size must be capable of being evaluated as a positive integer. If necessary, it is converted to an integer. For example, 5.2 is acceptable and is converted to 5. The number 0.2 is not acceptable, however, because it is converted to 0.

You can use the intrinsic function NWORKERS to help establish the distribution size (see Section 15.7.4).

15.7.2.2 SHARED, CONTEXT_SHARED, and PRIVATE Directives

The SHARED, CONTEXT_SHARED, and PRIVATE directives can be interspersed with declaration statements within program units in a program to be run in parallel. The functions of these directives are as follows:

- The SHARED directive causes memory locations for user-specified common blocks to be shared among all of the processes executing a parallel DO loop.
- The PRIVATE directive causes memory locations for user-specified variables and common blocks to be private to each of the processes executing iterations of a parallel DO loop.
- The CONTEXT_SHARED directive causes user-specified variables to be treated in memory as shared or private variables, depending on the context in which they are used.

The SHARED, CONTEXT_SHARED, and PRIVATE directives have no effect on code that is executing in a nonparallel context (that is, in code that is executed before entry into a parallel DO loop and after the completion of the loop). Note that any given common block should have the same attribute (shared or private) in all program units in a program to be run in parallel.

The SHARED, CONTEXT_SHARED, and PRIVATE directives have the following format:

```
column 1
|
CPAR$ PRIVATE name[,name]...
CPAR$ PRIVATE_ALL

CPAR$ SHARED common_name[,common_name]...
CPAR$ SHARED_ALL
CPAR$ CONTEXT_SHARED var_name[,var_name]...
CPAR$ CONTEXT_SHARED_ALL
```

PRIVATE_ALL
PRIVATE name[,name]...

Identifies those variables (scalars, arrays, and records) and common blocks that must have unique memory locations within each of the processes executing a parallel DO loop.

An attribute of private must be given to loop control variables of parallel DO loops. Variables that are always defined in each loop iteration before being used should also be declared as private.

If common blocks or variables are declared as private in any routine, they must be declared as private (or default to private) in all of the other routines that reference them.

Values for private variables and common blocks established before entry into a parallel DO loop should not be used inside the loop. Similarly, values for private variables and common blocks established inside a parallel DO loop should not be used outside the loop after its completion.

PRIVATE_ALL causes all variables and common blocks declared in a routine to be private—unless they are explicitly declared as shared. PRIVATE_ALL does not disallow the use of the SHARED and CONTEXT_SHARED directives; it merely establishes the default behavior for data sharing as PRIVATE, overriding the default behavior (SHARED_ALL and CONTEXT_SHARED_ALL) established by the /PARALLEL qualifier.

Commas are required between the names, and common block names must be enclosed by slashes (for example, /name/ or, for blank common, / /).

SHARED_ALL
SHARED common_name[,common_name]...

Identifies common blocks that are to be shared among all of the processes executing the compilation unit—in both parallel and nonparallel (serial) processing contexts. Specifying variables on a SHARED directive is disallowed.

Common blocks containing variables that meet the following criteria must be shared:

- Those defined before entry into a loop and used—without first being redefined—within any of the loop iterations.

- Those defined inside a parallel DO loop and used—without first being redefined—by code executed after completion of the loop.

SHARED_ALL does not disallow the use of the PRIVATE directive; it merely reinforces the default behavior for data sharing established by /PARALLEL. By default, all common blocks in a compilation unit compiled with the /PARALLEL qualifier are shared.

Common block names must be enclosed by slashes (for example, /name/ or, for blank common, / /). Commas are also required between the names.

CONTEXT_SHARED_ALL
CONTEXT_SHARED var_name[,var_name]...

Identifies those variables (scalars, arrays, and records) that are to reside in a shared memory location throughout any one invocation of a subprogram (subroutine or function), including any parallel loops contained with the subprogram. However, if a subprogram has several concurrent invocations (because it is called from within a parallel loop), each invocation will use different memory locations for these variables. (This context adjustment is handled automatically by the compiler and is not a programming consideration.) Specifying common blocks on a CONTEXT_SHARED directive is disallowed.

By default, all variables in a routine compiled with the /PARALLEL qualifier are context-shared.

Commas are required between variable names specified on a CONTEXT_SHARED directive.

The following restrictions apply to the use of SHARED, CONTEXT_SHARED, and PRIVATE directives:

- Arrays cannot be dimensioned within CONTEXT_SHARED and PRIVATE directives. For example, the array reference in the following directive would generate a compile-time error because of the dimensioning associated with ARRAY1:

```
CPAR$ PRIVATE ARRAY1(10,20),VAR1,/COMMON_1/
```

  The following statements show how to do this properly:

```
      REAL*4 ARRAY1(10,20)
CPAR$ PRIVATE ARRAY1,VAR1,/COMMON_1/
```

- Private variables and common blocks cannot be referenced in a SAVE statement.

- Common blocks cannot contain both shared and private variables. Any common block that has this property must be split into separate shared and private common blocks.
- Variables declared within a common block cannot be specified on a CONTEXT_SHARED or PRIVATE directive.
- Variables (scalars, arrays, and records) cannot be specified in both CONTEXT_SHARED and PRIVATE directives within the same program unit.
- Every program unit that references a common block must declare it as shared or private. All of the declarations must match. The declarations can be made explicitly by specifying directives or implicitly by taking the compiler defaults. The defaults are as follows:
  - In program units compiled with the /NOPARALLEL qualifier, all common blocks are private.
  - In program units compiled with the /PARALLEL qualifier, all common blocks are shared.

In general, ensure that the attributes and usage of variables outside parallel DO loops conform, as necessary, to constraints imposed by their attributes and their usage within the loop. For example, common blocks and variables must be declared to have the same memory allocation attributes (shared or private) in both parallel and nonparallel processing contexts, and private variables given values within a parallel DO loop cannot be used reliably after the completion of the loop without being redefined.

See Section 15.3.1 for information about when to use SHARED, CONTEXT_SHARED, and PRIVATE directives to resolve certain types of data dependence problems.

15.7.2.3 LOCKON and LOCKOFF Directives

The LOCKON and LOCKOFF directives can be used within a parallel DO loop to prevent multiple processes from executing selected statements in parallel. These directives force the multiple processes executing a parallel DO loop to execute selected statements serially. This can be useful when a statement (or set of statements) creates an unacceptable data dependence problem that cannot be resolved by other means.

The ability to set locks becomes useful when variables or common blocks have conflicting status requirements. For example, they are involved in a data dependence that crosses loop iterations (which requires them to be private) and they are used in code outside the loop (which requires them to be shared).

The LOCKON and LOCKOFF directives have the following forms:

```
column 1
|
CPAR$ LOCKON lock-variable
CPAR$ LOCKOFF lock-variable
```

A lock variable can be a variable or a dummy argument. It must have a data type of LOGICAL*4. For all of the processes to have access to it, it must have a memory allocation attribute of shared. A lock variable can be established in two ways:

- It can be declared in a shared common block.
- It can be declared as a context-shared variable within the routine containing the parallel DO loop.

The lock is in effect when the lock variable has a value of .TRUE. and unlocked when the lock variable has a value of .FALSE.

The LOCKON and LOCKOFF directives perform the following operations:

LOCKON	Waits, if necessary, for the lock variable to become .FALSE., then sets it to .TRUE. (that is, locks the lock), and then proceeds.
LOCKOFF	Sets the lock variable to .FALSE. (that is, unlocks the lock).

These directives use the VAX interlocked instructions to guarantee proper synchronization on a multiprocessor. Do not use any other statements to modify the lock variable while another process may be executing a LOCKON or LOCKOFF directive.

See Section 15.3.3 for examples of how locks are used in parallel DO loops.

15.7.3 Customizing the Parallel-Processing Run-Time Environment

To allow the user to tune the parallel-processing run-time environment in which a program is executed, the VAX FORTRAN Run-Time Library defines the following logical names:

Logical Name	Use
FOR$PROCESSES	Controls the number of processes used to execute a VAX FORTRAN program in parallel (32 maximum)
FOR$SPIN_WAIT FOR$STALL_WAIT	Control CPU usage when waiting, for work or synchronization, in a VAX FORTRAN program executing in parallel

You can define your own values for the logical names using the DCL commands DEFINE or ASSIGN. For example:

```
$ DEFINE FOR$PROCESSES 4
```

The values defined when a program starts parallel execution remain in effect until execution is completed.

Controlling the Number of Processes—FOR$PROCESSES

The logical name FOR$PROCESSES defines the number of processes to be used when executing a VAX FORTRAN program in parallel. To define FOR$PROCESSES, you must specify a nonzero, positive number. The maximum number is 32. If you do not define a value for FOR$PROCESSES, a default value equal to the number of processors that are currently active on the system is used.

Being able to adjust the number of processes can be helpful for a variety of reasons:

- It enables you to execute your parallel program in a single process. This allows you to debug the logic within your parallel DO loops as they execute in a serial, nonparallel fashion. (Note that running a program with parallel DO loops in one process (serially) does not reduce the initialization overhead associated with the parallel DO loops.)
- It enables you to compare the performance impact of executing a parallel program with a varying number of processes.

- It enables you to gauge the tradeoffs between increasing system overhead and increasing execution time. For example, in a time-sharing environment, you may find it advisable to reduce the number of processes in order to minimize contention for system resources.

FOR$PROCESSES is useful when you are executing a VAX FORTRAN program in parallel on a multiprocessor with more than two processors and you do not want to contend for the use of all of the available processors.

Controlling Internal Spin Waits—FOR$SPIN_WAIT

The logical name FOR$SPIN_WAIT allows you to tune the synchronization method used when a VAX FORTRAN program runs in parallel.

The synchronization methods used by a program running in parallel in a multitasking environment must deal with two conflicting goals:

- To respond as quickly as possible to a synchronization flag. The common way to accomplish this is to repeatedly test for an appropriate flag in a shared storage location. Doing this test in a tight loop ensures a quick response when the flag is reset. This solution, however, conflicts with the second goal.
- To avoid wasting valuable CPU cycles that might be used by another program.

Because of this conflict, a tradeoff must be made between the fastest response to the synchronization flags and fairness to other programs.

The FORTRAN Run-Time Library allows you to affect this tradeoff by defining a value for FOR$SPIN_WAIT. You can define it to be any nonnegative integer. This value specifies how many iterations of the spin-wait loop will execute before the executing process gives up the processor and allows VMS to schedule another process.

- A value of 0 is a special case that tells the run-time support to use the fastest synchronization at the expense of wasted CPU cycles. This value is appropriate for running a program in parallel on a system that is dedicated to running that single program.
- Other positive values tell the run-time support to use more or fewer spin-wait iterations, with higher values indicating more iterations. Thus, a value of 1 ensures the least wasted cycles—at the cost of the slowest synchronization response.

It is usually not necessary to define this logical name. The default value (1000) established by the run-time system should be adequate for most programs.

Controlling the State of a Process—FOR$STALL_WAIT

When a subprocess is waiting to work on a parallel DO loop, it can be either in an active state on the system or in an inactive state. When a subprocess is inactive, it becomes less responsive because it has to become active again before it can respond to the parallel DO loop.

As a second level of control over the internal spin waits in the parallel processing environment, the logical name FOR$STALL_WAIT allows you to control the time that a subprocess stays active, or computable, on the system. To control how long it remains active, you define a value for the logical name FOR$STALL_WAIT. This nonnegative value specifies the number of times that the subprocess will give up the CPU before becoming inactive.

- A value of 0 tells the run-time support to always maintain the subprocess as active, thus being more responsive when a parallel DO loop becomes available. A value of 0 is appropriate for programs that contain mostly parallel DO loops.
- Other positive values tell the run-time support to stay active for a longer or shorter interval, with higher values directing it to stay active longer. Thus, a value of 1 ensures that a subprocess waiting for a parallel DO loop will stay active for the shortest time interval. A value of 1 is appropriate when the program has large segments of code before, after, or between parallel DO loops.

It is usually not necessary to define this logical name. The default value (10 times the number of subprocesses) established by the run-time system should be adequate for most programs.

15.7.4 NWORKERS Intrinsic Function

The intrinsic function NWORKERS requires no arguments and returns an INTEGER*4 value that represents the total number of processes executing an application. NWORKERS will be most useful in parallel applications for determining the size of the iteration segments to be executed in the parallel processes. For example:

```
CPAR$ DO_PARALLEL  (N+NWORKERS( )-1) / NWORKERS( )
      DO I = 1, N
          .
     .
     .
      ENDDO
```

In this example, the size of each iteration segment to be executed in the parallel processes would be the total number of iterations in the parallel DO loop divided by the number of available processes. (Specifying "(N+NWORKERS( )-1)" guarantees that you do not get an illegal segment value of 0.)

Using NWORKERS, you can adjust the size of the iteration segment automatically to the number of processors available on the system on which the application is run.

Appendix A

Working with the Multiprocess Debugging Configuration

This appendix assumes that you are familiar with the configuration of the VMS Debugger that supports single-process debugging (referred to in this appendix as the "default debugging configuration"). It covers only the extensions that are provided to support multiprocess debugging. (See Chapter 3 for information about how to use the debugger for single-process debugging.)

This appendix provides information in the following areas:

- Basic information that you need to perform multiprocess debugging (Section A.1)
- Supplemental information on more advanced concepts and usage than those described in Section A.1 (Section A.2)
- System management considerations associated with multiprocess debugging (Section A.3)

The information in this appendix is oriented toward multiprocess debugging in general, not toward VAX FORTRAN parallel-processing debugging. See Section 15.5 for an overview specifically oriented toward VAX FORTRAN debugging.

A.1 Getting Started

This section gives a quick overview of the multiprocess debugging environment, by running through the basic steps and commands. Later sections are referenced for additional details. See the debugger's on-line HELP for complete details on commands.

A.1.1 Establishing a Multiprocess Debugging Configuration

Before invoking the debugger, enter the following command to establish a multiprocess configuration:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
```

This command establishes a multiprocess configuration for the job tree in which the command was issued. As a result, once a debugging session is started, any debuggable image running in the same job tree can be controlled from that one session. (An image is debuggable if it has been compiled and linked with the /DEBUG qualifier.)

A.1.2 Invoking the Debugger

This section explains the usual way of starting a multiprocess debugging session. See Section A.2.3 for additional techniques for invoking the debugger (for example, using the CONNECT command or a CTRL/Y - DEBUG sequence).

You typically initiate the execution of a multiprocess program by running the main image in the main (master) process. Once the main image is running in the main process, the program will spawn one or more subprocesses to run additional images by issuing a LIB$SPAWN run-time library call or a $CREPRC system service call. (Note: VAX FORTRAN performs this step during the initialization phase.)

If the main image is debuggable, the debugger is invoked when you run the image. For example:

```
$ RUN MAIN_PROG
          VAX DEBUG Version X5.0-3 MP

%DEBUG-I-INITIAL, language is FORTRAN, module set to MAIN_PROG
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
predefined trace on activation at routine MAIN_PROG in %PROCESS_NUMBER 1
DBG_1>
```

As with a one-process program, the debugger displays its banner and prompt just prior to the start of execution of the main image. However, note two differences—the "predefined trace on . . . " message and the debugger prompt.

In a multiprocess configuration, the debugger traces each new process that is brought under control. In this case, the debugger traces the first process, which runs the main image of the program. (%PROCESS_NUMBER is a built-in symbol that identifies a process number, just as %LINE identifies a line number.)

The significance of the prompt suffix (_1) is explained in the next section.

A.1.3 The Visible Process and Process-Specific Commands

The previous example shows that the debugger prompt in a multiprocess debugging configuration is different from that found in the default configuration.

In a multiprocess configuration, "dynamic prompt setting" is enabled (SET PROMPT/SUFFIX=PROCESS_NUMBER) by default. Therefore, the prompt has a process-specific suffix that indicates the process number of the *visible* process. The debugger assigns a process number sequentially, starting with process 1, to each process that comes under the control of a given debugging session.

The visible process is the process that is the default context for issuing process-specific commands. Process-specific commands are those that start execution (STEP, GO, and so on) and those used for looking up symbols, setting breakpoints, looking at the call stack and registers, and so on. Commands that are not process specific are those that do not depend on the mapping of virtual memory but, rather, affect the entire debugging environment (for example, keypad mode and screen mode commands).

Unless dynamic prompt setting is disabled (SET PROMPT/NOSUFFIX), the debugger prompt suffix always identifies the visible process (for example, DBG_1>). The SET PROMPT command provides several options for tailoring the prefix and suffix of the prompt string to your needs.

A.1.4 Obtaining Information About Processes

Use the SHOW PROCESS command to obtain information about processes that are currently under control of your debugging session. By default, SHOW PROCESS displays one line of information about the visible process. The following example shows the kind of information displayed immediately after you invoke the debugger:

```
DBG_1> SHOW PROCESS
 Number  Name         Hold  State          Current PC
*    1 JONES                activated      MAIN_PROG\%LINE 2
DBG_1>
```

A one-line SHOW PROCESS display provides the following information about each process specified:

- The process number assigned by the debugger. In this case, the process number is 1 because this is the first process known to the debugger. The asterisk in the leftmost column (*) marks the visible process.
- The VMS process name. In this case, the VMS process name is JONES.
- Whether the process has been placed on hold with a SET PROCESS /HOLD command, as explained in Section A.1.7.2. In this case, the process has not been placed on hold.
- The current debugging state for that process. A process is in the "activated" state when it is first brought under debugger control (that is, before it has executed any part of the program under debugger control). Table A–1 summarizes the possible debugging states that may appear in the state column.
- The location (symbolized, if possible) where execution of the image is suspended in that process. In this case, the image has not started execution.

Table A–1: Debugging States

Activated	The image and its process have just been brought under debugger control, either through a DCL RUN /DEBUG command, a debugger CONNECT command, a CTRL/Y - DEBUG sequence, or by the program signaling SS$DEBUG while it was not under debugger control.
Break[1]	A breakpoint was triggered.
Interrupted	Execution was interrupted in that process, either because execution was suspended in some other process or because the user interrupted program execution with the abort-key sequence (CTRL/C, by default).
Step[1]	A STEP command has completed.
Terminated	The image has terminated execution but the process is still under debugger control. Therefore, you can obtain information about the image and its process.
Trace[1]	A tracepoint was triggered.
Unhandled exception	An unhandled exception was encountered.
Watch of	A watchpoint was triggered.

[1]See the SHOW PROCESS command in the command dictionary for a list of additional states.

The SHOW PROCESS/ALL command provides information about all processes that are currently under debugger control. In the case of the previous example, a SHOW PROCESS/ALL command would show only process 1. The SHOW PROCESS/FULL command provides additional details about processes.

A.1.5 Bringing a Spawned Process Under Debugger Control

This section describes, in general, how the debugger interacts with spawned processes.

NOTE

Most of the information in this section is not pertinent to the debugging of parallel DO loops. The connect operations described in this section are performed for you automatically during the initialization phase.

To illustrate the interaction, assume that you are entering a few STEP commands and, in the middle of a step, MAIN_PROG spawns a process to run a debuggable image called TEST.

Because DBG$PROCESS has the value MULTIPROCESS, the spawned process is now requesting to connect to the current debugging session, and image TEST is suspended at the start of execution.

While the spawned process is waiting to be connected, it is not yet known to the debugger and cannot be identified in a SHOW PROCESS/ALL display. You can bring the process under debugger control using either of the following methods:

- Enter a command, such as STEP, that starts execution.
- Enter the CONNECT command without specifying a parameter. The CONNECT command is preferable in those cases when you do not want the program to execute any further.

The following example shows how to use the CONNECT command:

```
DBG_1> STEP
stepped to MAIN_PROG\%LINE 18 in %PROCESS_NUMBER 1
18:        LIB$SPAWN("RUN/DEBUG TEST")
DBG_1> STEP
stepped to MAIN_PROG\%LINE 21 in %PROCESS_NUMBER 1
21:          X = 7
DBG_1> CONNECT
predefined trace on activation at routine TEST in %PROCESS_NUMBER 2
DBG_1>
```

In this example, the second STEP command takes you past the LIB$SPAWN call that spawns the process. The CONNECT command brings the waiting process under debugger control. After entering the CONNECT command, you may need to wait a moment for the process to connect. The "predefined trace on . . . " message, as explained in Section A.1.2, indicates that the debugger has taken control of a new process and identifies that process as process 2, the second process known to the debugger in this session.

A SHOW PROCESS/ALL command, entered at this point, identifies the debugging state for each process and the location at which execution is suspended:

```
DBG_1> SHOW PROCESS/ALL
 Number  Name          Hold  State          Current PC
*    1 JONES                 step           MAIN_PROG\%LINE 21
     2 JONES_1               activated      TEST\%LINE 1+2
DBG_1>
```

Note that the CONNECT command brings any processes that are waiting to be connected to the debugger under debugger control. If no processes are waiting, you can press CTRL/C to abort the CONNECT command and display the debugger prompt.

A.1.6 Broadcasting Commands to Selected Processes

By default, process-specific commands are executed in the context of the visible process. The DO command enables you to execute commands in the context of one or more processes that are currently under debugger control. This capability is referred to as "broadcasting" commands to processes.

Use the DO command without a qualifier to execute commands in the context of all of the processes. For example, the following command executes the SHOW CALLS command for all processes that are currently under debugger control (processes 1 and 2, in this case):

```
DBG_1> DO (SHOW CALLS)
For %PROCESS_NUMBER 1
   module name      routine name            line       rel PC    abs PC
  *MAIN_PROG        MAIN_PROG                 21     0000001E  0000041E
For %PROCESS_NUMBER 2
   module name      routine name            line       rel PC    abs PC
   TEST             TEST                     1+2     0000000B  0000040B
```

Use the DO command with the /PROCESS= qualifier to execute commands in the context of selected processes. For example, the following command executes the commands SET MODULE START and EXAMINE X in the context of process 2 (see Section A.2.1 for information on how to specify processes in debugger commands):

```
DBG_1> DO/PROCESS=(%PROC 2) (SET MODULE START; EXAMINE X)
```

A.1.7 Controlling Execution

Program execution in a multiprocess debugging environment follows these conventions:

- When you enter a command that starts program execution, such as STEP or GO, the command is executed in the context of the visible process. However, images in any other unheld processes (processes that have not been placed on hold with a SET PROCESS/HOLD command) are also allowed to execute. Similarly, if you use the DO command to broadcast a command to start execution in one or more processes, the command is executed in the context of each specified unheld process, but images in any other unheld processes are also allowed to execute. In all cases, a hold condition is ignored in the visible process. (See Section A.1.7.2 for additional information about the behavior of processes when on hold.)
- Once execution is started, the way in which it continues depends on whether the command SET MODE [NO]INTERRUPT was entered. By default (SET MODE INTERRUPT), execution continues until it is suspended in any process. At that point, execution is interrupted in any other processes that were executing images, and the debugger prompts for input.

These concepts are illustrated next by continuing with the example in Section A.1.5 that shows the two STEP commands.

In that example, the "stepped to..." messages indicate that both commands are executed in the context of process 1, the visible process. The second STEP command spawns process 2. The SHOW PROCESS/ALL example of Section A.1.5 indicates that execution in processes 1 and 2 is suspended at MAIN_PROG\%LINE 21 and TEST\%LINE 1+2, respectively.

At this point, entering another STEP command followed by SHOW PROCESS/ALL results in the following display:

```
DBG_1> STEP
stepped to MAIN_PROG\%LINE 23 in %PROCESS_NUMBER 1
23:         Y = 15
DBG_1> SHOW PROCESS/ALL
 Number  Name          Hold  State           Current PC
*    1 JONES                 step            MAIN_PROG\%LINE 23
     2 JONES_1               interrupted     TEST\%LINE 3+1
DBG_1>
```

The STEP command is executed in the context of process 1, the visible process. After the STEP, execution in process 1 is suspended at MAIN_PROG\%LINE 23. However, the STEP command also causes execution to start in process 2. The completion of the STEP in process 1 causes execution in process 2 to be interrupted at TEST\%LINE 3+1.

Section A.1.7.1 describes another mode of execution, which is provided by the command SET MODE NOINTERRUPT.

A.1.7.1 Controlling Execution with SET MODE NOINTERRUPT

SET MODE NOINTERRUPT allows execution to continue without interruption in other processes when it is suspended in some process. This is especially useful if, for example, you want to broadcast a STEP command to several processes with the DO command and complete execution of the STEP in all these processes. For example:

```
DBG_1> SET MODE NOINTERRUPT
DBG_1> DO (STEP)
```

In this example, the DO command executes the STEP command in the context of all processes. The visible process and any other unheld processes start execution. Because the command SET MODE NOINTERRUPT was entered, the prompt is displayed only after the STEP has completed (or execution has been otherwise suspended at a breakpoint or watchpoint) in all processes that were executing.

When SET MODE NOINTERRUPT is in effect, as long as execution continues in any process, the debugger does not prompt for input. In such cases, use CTRL/C to interrupt all processes and display the prompt.

A.1.7.2 Putting Selected Processes on Hold

As indicated in the preceding sections, a command that starts execution is executed in the context of the visible process, but it also causes execution to start in other processes. If you want to inhibit execution in a process, put it on hold. For example, the following SET PROCESS/HOLD command puts process 2 on hold. The subsequent STEP command is executed in the context of process 1, the visible process. Execution also starts in any other processes that are not on hold, but not in process 2:

```
DBG_1> SET PROCESS/HOLD %PROC 2
DBG_1> STEP
```

A SHOW PROCESS display indicates whether a process is on hold. For example:

```
DBG_1> SHOW PROCESS/ALL
 Number  Name          Hold  State            Current PC
*    1 JONES                 step             MAIN_PROG\%LINE 24
     2 JONES_1         HOLD  interrupted      TEST\%LINE 3+1
DBG_1>
```

To "unhold" a process, enter the command SET PROCESS/NOHOLD, specifying the process that you want to release from the hold condition.

Note that a hold condition is ignored in the visible process. Therefore, the command SET PROCESS/HOLD/ALL is a convenient way to confine execution to the visible process. In the following example, execution starts only in the visible process:

```
DBG_1> SET PROCESS/HOLD/ALL
DBG_1> STEP
```

This feature is useful if, for example, you want to use the CALL command to execute a dump routine that is not part of the execution stream of your program.

The preceding discussions also apply if you use the DO command to broadcast a GO, STEP, or CALL command to several processes. The GO, STEP or CALL command is executed in the context of each specified unheld process, and execution also starts in any other unheld process. The following example shows the execution behavior when all processes are placed on hold and commands are broadcast to all processes. Execution starts only in the visible process (process 1, in this example):

```
DBG_1> SET PROCESS/HOLD/ALL
DBG_1> DO (EXAMINE X; STEP)
For %PROCESS_NUMBER 1
  MAIN_PROG\X:    78
For %PROCESS_NUMBER 2
  TEST\X:    29
stepped to MAIN_PROG\%LINE 26 in %PROCESS_NUMBER 1
26:     K = K + 1
DBG_1>
```

A.1.8 Changing the Visible Process

Use the SET PROCESS command (with the default /VISIBLE qualifier) to establish another process as the visible process. For example, the following command makes process 2 the visible process:

```
DBG_1> SET PROCESS %PROC 2
DBG_2>
```

In this example, because dynamic prompt setting is enabled by default, the SET PROCESS command has also caused the prompt string suffix to change. It now indicates that process 2 is the visible process. All process-specific commands are now executed in the context of process 2. For example, a SHOW CALLS command would display the call stack for the image running in process 2.

A.1.9 Dynamic Process Setting

By default, "dynamic process setting" is enabled (SET PROCESS /DYNAMIC). As a result, whenever the debugger suspends program execution, the process in which execution is suspended becomes the visible process automatically. Dynamic process setting occurs in the following situations: when a breakpoint or watchpoint is triggered, at an exception condition, on the completion of a STEP command, or when the last process performs an image exit.

When dynamic process setting is disabled (/NODYNAMIC), the visible process remains unchanged until you specify another process with the SET PROCESS/VISIBLE command.

Dynamic process setting is illustrated in the following example, which also illustrates dynamic prompt setting:

```
DBG_1> SHOW PROCESS/ALL
 Number  Name           Hold  State            Current PC
*     1 JONES                 step             MAIN_PROG\%LINE 22
      2 JONES_1               interrupted      TEST\%LINE 4
DBG_1> DO/PROCESS=(%PROC 2) (SET BREAK %LINE 11)
DBG_1> GO
      .
      .
      .
break at TEST\%LINE 11 in %PROCESS_NUMBER 2
DBG_2> SHOW PROCESS/ALL
 Number  Name           Hold  State            Current PC
      1 JONES                 interrupted      MAIN_PROG\%LINE 28
*     2 JONES_1               break            TEST\%LINE 11
DBG_2>
```

In this example, process 1 is initially the visible process, as indicated by the prompt and the SHOW PROCESS display. The DO command sets a breakpoint in the context of process 2. Execution is resumed with the GO command and is suspended at the breakpoint in process 2. Process 2 is now the visible process, as indicated by the prompt and the SHOW PROCESS display.

If you had entered the command SET MODE NOINTERRUPT and then had started execution in several processes with the DO command, the prompt would not be displayed until after execution was suspended in all processes. In this case, the visible process remains unchanged, unless the last process performs an image exit (and thereby becomes the visible process).

A.1.10 Monitoring the Termination of Images

When the main image of a process runs to completion, the process goes into the "terminated" debugging state. This condition is traced by default, as if you had entered the command SET TRACE/TERMINATING.

When a process is in the terminated state, it is still known to the debugger and appears in a SHOW PROCESS/ALL display. You can enter commands to examine variables, and so on.

When the last image of the program exits, the debugger gains control and displays its prompt.

A.1.11 Terminating the Debugging Session

To terminate the entire debugging session, use the EXIT or QUIT command without specifying any parameters. When you do not specify parameters, the behavior of EXIT and QUIT is analogous to their behavior for the default debugging configuration. (QUIT does not execute any user-declared exit handlers.)

A.1.12 Releasing Selected Processes from Debugger Control

To release selected processes from debugger control without terminating the debugging session, use the EXIT or QUIT command, specifying one or more process specifications as parameters. For example, the following command terminates the image running in process 2, and releases the process from debugger control:

```
DBG_3> EXIT %PROC 2
DBG_3>
```

Subsequently, process 2 does not appear in a SHOW PROCESS display. See the command dictionary for complete details on the EXIT and QUIT commands.

A.1.13 Aborting Debugger Commands and Interrupting Program Execution

Use CTRL/C (not CTRL/Y) to abort the execution of a debugger command or to interrupt program execution. This is useful if a command takes a long time to complete or your program loops. Control is returned to the debugger rather than to the DCL command interpreter. For example:

```
DBG_1> GO
    .
    .
    .
CTRL/C
%DEBUG-W-ABORTED, command aborted by user request
DBG_1> EXAMINE/BYTE 1000:101000  !should have typed 1000:1010
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0
CTRL/C
%DEBUG-W-ABORTED, command aborted by user request
DBG_1>
```

Pressing CTRL/C interrupts execution in every process that is currently running an image. This is indicated as an "interrupted" state in a SHOW PROCESS display. Pressing CTRL/C also interrupts any debugger command that is currently executing.

If your program already has a CTRL/C AST service routine enabled, use the SET ABORT_KEY command to assign the debugger's abort function to another CTRL-key sequence. For example:

```
DBG_1> SET ABORT_KEY = CTRL_P
DBG_1> GO
    .
    .
    .
CTRL/P
%DEBUG-W-ABORTED, command aborted by user request
DBG_1> EXAMINE/BYTE 1000:101000  !should have typed 1000:1010
1000: 0
1004: 0
1008: 0
1012: 0
1016: 0
CTRL/P
%DEBUG-W-ABORTED, command aborted by user request
DBG_1>
```

Note, however, that many CTRL-key sequences have VMS predefined functions, and the SET ABORT_KEY command enables you to override such definitions within the debugging session (see the *VMS DCL Concepts Manual*). Some of the CTRL-key characters not used by the VMS operating system are G, K, N, and P.

A.2 Supplemental Information

Section A.1 describes the fundamental operations associated with multiprocess debugging. This section provides details on advanced concepts and usages that relate to multiprocess debugging.

A.2.1 Specifying Processes in Debugger Commands

When specifying processes in debugger commands, you can use any of the forms listed in Table A–2, except when specifying processes with the CONNECT command.

The CONNECT command is used to bring a process that is not yet known to the debugger under debugger control. Therefore, when specifying a process with CONNECT, you can use only its VMS process name or VMS process identification number (PID). You cannot use its debugger-assigned process number or any of the process built-in symbols (for example, %NEXT_PROCESS). (As noted earlier in this appendix, the CONNECT command is not used in the debugging of VAX FORTRAN parallel DO loops.)

Table A–2: Process Specifications

[%PROCESS_NAME] process-name	The VMS process name, if that name contains no spaces or lowercase characters[1].
[%PROCESS_NAME] "process-name"	The VMS process name, if that name contains spaces or lowercase characters. You can also use apostrophes (') instead of quotation marks (").
%PROCESS_PID process_id	The VMS process identification number (PID, a hexadecimal number).

[1]The process name can include the wildcard character (*)

Table A-2 (Cont.): Process Specifications

%PROCESS_NUMBER process-number (or %PROC process-number)	The number assigned to a process when it comes under debugger control. A new number is assigned sequentially, starting with 1, to each process. If a process is released from debugger control (with the EXIT or QUIT command), the number is not reused during the debugging session. Process numbers appear in a SHOW PROCESS display. Processes are ordered in a circular list so they can be indexed with the built-in symbols %PREVIOUS_PROCESS and %NEXT_PROCESS.
process-group-name	A symbol defined with the DEFINE /PROCESS_GROUP command to represent a group of processes.
%NEXT_PROCESS	The next process after the visible process in the debugger's circular process list.
%PREVIOUS_PROCESS	The process previous to the visible process in the debugger's circular process list.
%VISIBLE_PROCESS	The process whose stack, register set, and images are the current context for looking up symbols, register values, routine calls, breakpoints, and so on.

You can omit the %PROCESS_NAME built-in symbol when entering commands. For example:

```
DBG_2> SHOW PROCESS %PROC 2, JONES_3
```

You can define a symbol to represent a group of processes (DEFINE /PROCESS_GROUP). This enables you to enter commands in abbreviated form. For example:

```
DBG_1> DEFINE/PROCESS_GROUP SERVERS=FILE_SERVER, NETWORK_SERVER
DBG_1> SHOW PROCESS SERVERS
 Number  Name          Hold  State            Current PC
*    1 FILE_SERVER           step             FS_PROG\%LINE 37
     2 NETWORK_SERVER        break            NET_PROG\%LINE 24
DBG_1>
```

The built-in symbols %VISIBLE_PROCESS, %NEXT_PROCESS, and %PREVIOUS_PROCESS are useful in control structures (IF, WHILE, REPEAT, and so on) and in command procedures.

.2.2 Monitoring Process Activation and Termination

By default, a tracepoint is triggered when a process comes under debugger control and when it performs an image exit. These predefined tracepoints are equivalent to those resulting from entering the commands SET TRACE/ACTIVATING and SET TRACE/TERMINATING, respectively. You can set breakpoints on these events by means of the SET BREAK /ACTIVATING and SET BREAK/TERMINATING commands.

To cancel the predefined tracepoints, use the CANCEL TRACE /PREDEFINED command with the /ACTIVATING and /TERMINATING qualifiers. To cancel any user-defined activation and termination breakpoints, use the CANCEL BREAK command with the /ACTIVATING and /TERMINATING qualifiers (the /USER qualifier is assumed by default when canceling breakpoints or tracepoints).

The debugger prompt is displayed when the first process comes under debugger control. This enables you to enter commands before the main image has started execution, just as with a one-process program.

Also, the debugger prompt is displayed when the last process performs an image exit. This enables you to enter commands after the program has completed execution, just as with a one-process program.

.2.3 Interrupting the Execution of an Image to Connect It to the Debugger

You can interrupt a debuggable image that is running without debugger control in a process and connect it to the debugger.

There are two general scenarios:

- To start a new debugging session, use the CTRL/Y - DEBUG sequence from DCL level.
- To interrupt the image and connect it to an existing debugging session, use the CONNECT command.

A.2.3.1 Using the CTRL/Y - DEBUG Sequence to Invoke the Debugger

You use the CTRL/Y - DEBUG sequence with the multiprocess debugging configuration exactly as with the default configuration. That is, run the image from DCL level with the RUN/NODEBUG command, then press CTRL/Y to interrupt the image. The DEBUG command causes the debugger to be invoked.

The following example shows how you might start a new debugging session:

```
$ DEFINE/JOB DBG$PROCESS MULTIPROCESS
$ RUN/NODEBUG PROG2
    .
    .
    .
CTRL/Y
Interrupt
$ DEBUG
          VAX DEBUG Version X5.0-3 MP

%DEBUG-I-INITIAL, language is FORTRAN, module set to SUB4
trace on activation at SUB4\%LINE 12 in %PROCESS_NUMBER 1
DBG_1>
```

In this example, the DEFINE/JOB command establishes a multiprocess debugging configuration. The RUN/NODEBUG command starts the execution of image PROG2 without debugger control. The CTRL/Y - DEBUG sequence interrupts execution and invokes the debugger.

The VAX DEBUG banner indicates that a new debugging session has been started. The process-specific prompt (DBG_1>) indicates that this is a multiprocess configuration and that execution is suspended in process 1, which is running image PROG2.

The activation tracepoint identifies the location at which execution was interrupted (and at which the debugger took control of the process). You can also use the SHOW CALLS command to display the call stack at that location.

After the debugger has been invoked, you can use the CONNECT command to bring other processes under debugger control. In the previous example, you could use the CONNECT command to bring processes under debugger control that were created by PROG2 before you interrupted its execution (see Section A.2.3.2).

When using the CTRL/Y - DEBUG sequence, if a multiprocess debugging session already exists in the same job tree as the image that is interrupted, the image connects to that particular session. In this case, because a new session is not started, the VAX DEBUG banner is not displayed when the debugger takes control. This situation could occur if, for example, you entered a SPAWN/NOWAIT command from the session, started execution with a RUN/NODEBUG command, and then entered a CTRL/Y - DEBUG sequence.

A.2.3.2 Using the CONNECT Command to Interrupt an Image

The CONNECT command, used without a parameter, was introduced in Section A.1.5. (As noted in that section, the CONNECT command is not used in the debugging of VAX FORTRAN parallel DO loops.)

When used with a parameter, the CONNECT command enables you to interrupt a debuggable image that is running without debugger control and bring it under control of your current debugging session.

The image may have been activated as follows:

- By your program issuing a LIB$SPAWN run-time library call or a $CREPRC system service call to spawn a process and run an image without debugger control
- By starting execution with a RUN/NODEBUG command entered at DCL level

In the following example, the CONNECT command causes the image running in process JONES_3 to be interrupted and to come under control of the current debugging session. Process JONES_3 must be in the same job tree as the session.

```
DBG_1> CONNECT JONES_3
```

Note that a process is not identified by a debugger process number until it is connected to a debugging session. Therefore, when specifying a process with the CONNECT command, you can use only its VMS process name or VMS process identification number (PID).

The effect of the CONNECT command is equivalent to attaching to a process from a debugging session and then entering the sequence CTRL/Y - DEBUG to interrupt the running image and invoke the debugger. However, the CONNECT command is simpler for you to enter and also enables you to interrupt a process to which you cannot attach.

A.2.4 Screen Mode Features for Multiprocess Debugging

Screen mode displays, whether predefined or user defined, are associated with the visible process, by default. For example, SRC shows the source code where execution is suspended in the visible process, OUT shows the output of commands executed in the context of the visible process, and so on.

By using the /PROCESS qualifier with the SET DISPLAY and DISPLAY commands you can create process-specific displays or make existing displays process-specific, respectively. The contents of a process-specific display are generated and modified in the context of that process. You can make any display to be process specific except for the PROMPT display. For example, the following command creates the automatically updated source display SRC_3, which shows the source code where execution is suspended in process 3:

```
DBG_2> SET DISPLAY/PROCESS=(%PROC 3) SRC_3 -
_DBG_2> AT RS23 SOURCE (EXAM/SOURCE .%SOURCE_SCOPE\%PC)
```

You assign attributes to process-specific displays in the same way you assign them to displays that are not process specific. For example, the following command makes display SRC_3 the current scrolling and source display — that is, it causes the output of SCROLL, TYPE, and EXAMINE /SOURCE commands to be directed at SRC_3:

```
DBG_2> SELECT/SCROLL/SOURCE SRC_3
```

If you enter a DISPLAY/PROCESS or SET DISPLAY/PROCESS command without specifying a process, the specified display is then specific to the process that was the visible process when you entered the command. For example, the following command makes OUT_X specific to process 2:

```
DBG_2> DISPLAY/PROCESS OUT_X
```

The /SUFFIX qualifier appends a process identifying suffix that denotes the visible process to a display name. This qualifier can be used directly after a display name in any command that specifies a display (for example, SET DISPLAY, EXTRACT, SAVE). It is especially useful within command procedures, in conjunction with display definitions or with key definitions that are bound to display definitions.

In a multiprocess configuration, the predefined tracepoint on process activation automatically creates a new source display and a new instruction display for each new process that comes under debugger control. The displays have the names SRC_*n* and INST_*n*, respectively, where *n* is the process number. These processes are initially marked as removed.

They are automatically canceled by the predefined tracepoint on process termination.

Several predefined keypad key sequences enable you to configure your screen with the process-specific source and instruction displays that are created automatically when a process is activated. Table A–3 identifies the keypad keys and describes their general effects. The table also describes any changes to the keypad keys from previous versions of the debugger. Use the SHOW KEY command to determine the exact commands issued by these key combinations.

Table A–3: Changed and New Keypad Key Functions

Key	State	Command Invoked or Function
COMMA	GOLD	SELECT/SOURCE %NEXT_SOURCE. Selects the next source display in the display list as the current source display. This function was previously assigned to KP3 in the BLUE state.
KP9	GOLD	SET PROCESS/VISIBLE %NEXT_PROCESS. Makes the next process in the process list the visible process.
KP9	BLUE	Displays two predefined process-specific source displays, SRC_*n*. These are located at Q1 and Q2, respectively, for the visible process and for the next process on the process list.
KP7	BLUE	Displays two sets of predefined process-specific source and instruction displays, SRC_*n* and INST_*n*. These consist of source and instruction displays for the visible process at Q1 and RQ1, respectively, and source and instruction displays for the next process on the process list at Q2 and RQ2, respectively.
KP3	BLUE	Displays three predefined process-specific source displays, SRC_*n*. These are located at S1, S2, and S3, respectively, for the previous, current (visible), and next process on the process list.
KP1	BLUE	Displays three sets of predefined process-specific source and instruction displays, SRC_*n* and INST_*n*. These consist of source and instruction displays for the visible process at S2 and RS2, respectively; source and instruction displays for the previous process on the process list at S1 and RS1, respectively; and source and instruction displays for the next process on the process list at S3 and RS3, respectively.

A.2.5 Setting Watchpoints in Global Sections

You can set watchpoints in global sections. A global section is a region of virtual memory that is shared among all processes of a multiprocess program. A watchpoint that is set on a location in a global section—a global section watchpoint—triggers when any process modifies the contents of that location.

Note that, when setting watchpoints on arrays or records, performance is improved if you specify individual elements rather than the entire structure with the SET WATCH command.

If you set a watchpoint on a location that is not yet mapped to a global section, the watchpoint is treated as a conventional static watchpoint. For example:

```
DBG_1> SET WATCH ARR(1)
DBG_1> SHOW WATCH
watchpoint of PPL3\ARR(1)
```

When ARR is subsequently mapped to a global section, the watchpoint is automatically treated as a global section watchpoint and an informational message is issued. For example:

```
DBG_1> GO
%DEBUG-I-WATVARNOWGBL, watched variable PPL3\ARR(1) has been remapped
        to a global section
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 2
     1:         PROGRAM PPL3
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 3
     1:         PROGRAM PPL3
watch of PPL3\ARR(1) at PPL3\%LINE 93 in %PROCESS_NUMBER 2
    93:             ARR(1) = INDEX
   old value: 0
   new value: 1
break at PPL3\%LINE 94 in %PROCESS_NUMBER 2
   94:              ARR(I) = I
```

Once the watched location is mapped to a global section, the watchpoint is visible from each process. For example:

```
DBG_2> DO (SHOW WATCH)
For %PROCESS_NUMBER 1
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
For %PROCESS_NUMBER 2
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
For %PROCESS_NUMBER 3
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
```

A.2.6 Compatibility of Multiprocess Commands with the Default Configuration

All of the commands, qualifiers, and built-in symbols that are provided for multiprocess debugging are also understood in the default debugging configuration and have analogous behaviors (where applicable). For example:

- The EXIT command without a parameter specified terminates a debugging session in both configurations.
- A DO command without the /PROCESS qualifier executes the commands specified in all processes.
- In the default configuration, the "visible" process is the process that runs the entire program. It is identified as process 1 in a SHOW PROCESS display.
- Built-in symbols such as %PROCESS_NUMBER and %VISIBLE_PROCESS are interpreted correctly in the default configuration.

This compatibility is especially useful because it allows command procedures used for multiprocess debugging to also be used for debugging programs that run in only one process.

A.3 System Management Considerations for Multiprocess Debugging

Several users debugging programs that occupy several processes can place a significant load on a system. This section describes the resources used by the multiprocess debugger and how to tune your system accordingly.

Note that the following discussion covers only the resources used by the debugger. You may also have to tune your system to support the execution of the multiprocess programs themselves (see Section 15.7.3).

A.3.1 User Quotas

Each user needs sufficient PRCLM quota to create an additional subprocess for the debugger, beyond the number of processes needed for the multiprocessing program. This quota may need to be increased to account for the debugger subprocess.

BYTLM, ENQLM, FILLM, and PGFLQUO are pooled quotas. These quotas may need to be increased to account for the debugger subprocess:

- Each user's ENQLM quota should be increased by at least the number of processes being debugged.
- Each user's PGFLQUO quota may need to be increased. If a user has insufficient PGFLQUO, the debugger may fail to activate, or produce "virtual memory exceeded" errors during execution.
- Each user's FILLM and BYTLM quotas may need to be increased. The debugger requires enough FILLM and BYTLM quotas to open each image file being debugged, the corresponding source files, and the DEBUG input, output, and log files. The DEBUG SET MAX_SOURCE_FILES command can be used to limit the number of source files kept open by the debugger at any one time.

A.3.2 System Resources

The kernel and main debugger communicate through global sections. The main debugger communicates with up to 8 kernel debuggers through a 65-page global section. Therefore, the SYSGEN parameters GBLPAGES and GBLSECTIONS may need to be increased. For example, if 10 users are using the debugger simultaneously, 10 global sections (GBLSECTIONS), using a total of 650 global pages (GBLPAGES), are required by the debugger.

Appendix B

Contents of the FORTRAN System Library FORSYSDEF

Table B–1 is a list of the modules contained in the FORTRAN system library FORSYSDEF. The modules consist of definitions, in FORTRAN source code, of related groups of system symbols that can be used in calling VMS system services. FORSYSDEF also contains modules that define the condition symbols and the entry points for Run-Time Library procedures.

Section 6.5.1 describes the procedure for accessing the modules listed below from a FORTRAN program. Section 9.1.2.3 describes condition values and symbols.

Table B–1: Contents of System Library FORSYSDEF

Module Name	Description
$ACCDEF	Accounting manager request type codes
$ACEDEF	Access control list entry structure definitions
$ACLDEF	Access control list interface definitions
$ACRDEF	Accounting record definitions
$ARGDEF	Argument descriptor for object language procedure records
$ARMDEF	Access rights mask longword definitions
$ATRDEF	File attribute list description—used to read and write file attributes
$BRKDEF	Breakthru ($BRKTHRU) system service input definitions

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$CHFDEF	Condition handling argument list offsets
$CHKPNTDEF	Create checkpointable processes flag definitions
$CHKPRO	Item definitions for $CHKPRO (check protection) system services
$CHPDEF	Check protection ($CHKPRO) system service definitions
$CLIDEF	Command language interface definitions—define the offset values for structures used to communicate information to CLI
$CLISERVDEF	CLI service request code definitions
$CLIVERBDEF	CLI generic verb codes definitions
$CLSDEF	Security classification mask block—contains security and integrity level categories for nondiscretionary access controls
$CQUALDEF	Common Qualifier package definitions
$CRDEF	Card reader status bits
$CREDEF	Create options table definitions for library facility
$CRFDEF	CRF$_INSRTREF argument list
$CRFMSG	Return status codes for cross reference program
$DCDEF	Device adapter, class, and type definitions
$DEVDEF	I/O device characteristics
$DIBDEF	Device information block definitions
$DMPDEF	Header block definitions of system dump file
$DMTDEF	Flag bits for the Dismount ($DISMOU) system service
$DSCDEF	Descriptor type definitions
$DSTDEF	Debug symbol table definitions
$DTKDEF	Definitions for RTL DECtalk management facility
$DTKMSG	Return status codes for RTL DECtalk management facility
$DVIDEF	Device and volume information data identifier definitions
$ENVDEF	Environment definitions in object file
$EOMDEF	End of module record in object/image files
$EOMWDEF	End of module record in object/image with word of psect value

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$EPMDEF	Global symbol definition record in object file—entry point definitions
$EPMWDEF	Global symbol definition record in object file—entry point definitions with word of psect value
$ERADEF	Erase type code definitions
$FABDEF	RMS File access block definitions
$FALDEF	Messages for the FAL (File Access Listener) facility
$FIBDEF	File identification block definitions
$FIDDEF	File ID structure
$FMLDEF	Formal argument definitions appended to procedure definitions in global symbol definition record in object file
$FORDEF	Condition symbols for FORTRAN-specific Run-Time Library
$FORIOSDEF	FORTRAN IOSTAT error numbers
$FSCNDEF	Descriptor codes for SYS$FILESCAN
$GPSDEF	Global symbol definition record in object file—psect definitions
$GSDEF	Global symbol definition (GSD) record in object file
$GSYDEF	Global symbol definition record in object file—symbol definitions
$HLPDEF	Data structures for help processing
$IACDEF	Image activation control flags
$IDCDEF	Object file IDENT consistency check structures
$IODEF	I/O function codes
$JPIDEF	Job/process information request type codes
$KGBDEF	Key grant block definitions—formats of records in rights database file
$LADEF	Laboratory peripheral accelerator device types
$LATDEF	Error messages for LAT facility
$LBRCTLTBL	Librarian control table definitions
$LBRDEF	Library type definitions
$LCKDEF	Lock manager definitions

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$LEPMDEF	Global symbol definition record in object file—module local entry point definitions
$LHIDEF	Library header information array offsets
$LIBCLIDEF	Definitions for LIB$ CLI callback procedures
$LIBDCFDEF	Definitions for LIB$DECODE_FAULT procedure
$LIBDEF	Condition symbols for the general utility library
$LIBDTDEF	Interface definiitons for LIB$DT (date/time) package
$LIBVMDEF	Interface definiitons for LIB$VM (virtual memory) package
$LKIDEF	Get lock information data identifier definitions
$LNKDEF	Linker option record definition in object file
$LNMDEF	Logical name flag definitions
$LPDEF	Line printer characteristic codes
$LPRODEF	Global symbol definition record in object file—module local procedure definition in object file
$LSDFDEF	Module local symbol definition in object file
$LSRFDEF	Module local symbol reference in object file
$LSYDEF	Module local symbol definition
$MHDEF	Module header record definition in object file
$MNTDEF	Flag bits and function codes for the MOUNT system service
$MSGDEF	Symbolic names to identify mailbox message senders
$MTDEF	Magnetic tape characteristic codes
$MTHDEF	Condition symbols from the mathematical procedures library
$NAMDEF	RMS name block field definitions
$NCSDEF	Interface definitions for National Character set package
$NSARECDEF	Security auditing record definitions
$OBJRECDEF	Object language record definition
$OPCDEF	Operator communication manager request type codes—return status codes
$OPCMSG	OPCOM message definitions
$OPDEF	Opcode values

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$OPRDEF	Operator communications message types and values
$OTSDEF	Language-independent support procedure (OTS$) return status codes
$PCCDEF	Printer/terminal carriage control specifiers
$PLVDEF	Privileged library vector definitions
$PQLDEF	Quota types for process creation quota list
$PR730DEF	VAX 11/730 processor specific definitions
$PR750DEF	VAX 11/750 processor specific definitions
$PR780DEF	VAX 11/780 processor specific definitions
$PRCDEF	Create process ($CREPRC) system service status flags and item codes
$PRDEF	Processor register definitions
$PRODEF	Global symbol definition record in object file—procedure definition
$PROWDEF	Global symbol definition record in object file—procedure definition with word of psect value
$PRTDEF	Protection field definitions
$PRVDEF	Privilege bit definitions
$PSLDEF	Processor status longword (PSL) mask and symbolic names for access modes
$PSWDEF	Processor status word mask and field definitions
$QUIDEF	Get queue information service definitions
$RABDEF	RMS record access block definitions
$RMEDEF	RMS escape definitions
$RMSDEF	RMS return status codes
$SBKDEF	Open file statistics block
$SCRDEF	Screen package interface definitions
$SDFDEF	Symbol record in object file
$SDFWDEF	Symbol record in object file with word of psect value
$SECDEF	Attribute flags for private/global section creation and mapping

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$SFDEF	Stack frame offset definitions
$SGPSDEF	Global symbol definition record in object file—P-section definition in shareable image
$SHRDEF	Definitions for shared messages
$SJCDEF	Send to job controller service definitions
$SMGDEF	Definitions for RTL screen management
$SMGMSG	Messages for the Screen Management facility
$SMGTRMPTR	Terminal capability pointers for RTL SMG$ facility
$SMRDEF	Define symbiont manager request codes
$SORDEF	Messages for the Sort/Merge facility
$SRFDEF	Global symbol definition record in object file—symbol reference definitions
$SRMDEF	SRM hardware symbol definitions
$SSDEF	System service failure and status codes
$STRDEF	String manipulation procedures (STR$) return status codes
$STSDEF	Status codes and error codes
$SYIDEF	Get system information data identifier definitions
$SYSSRVNAM	System service entry point descriptions
$TIRDEF	Text information and relocation record in object file
$TPADEF	TPARSE control block
$TRMDEF	Define symbols to the item list QIO format
$TT2DEF	Terminal special symbols
$TTDEF	Terminal device characteristic codes
$UICDEF	Format of user identification code (UIC)
$USGDEF	Disk usage accounting file produced by ANALYZE/DISK_STRUCTURE
$XABALLDEF	Allocation XAB definitions
$XABCXFDEF	RMS context XAB associated with FAB
$XABCXRDEF	RMS context XAB associated with RAB
$XABDATDEF	Date/time XAB definitions

Table B–1 (Cont.): Contents of System Library FORSYSDEF

Module Name	Description
$XABDEF	Definitions for all XABs
$XABFHCDEF	File header characteristics XAB definitions
$XABJNLDEF	Journal XAB definitions
$XABKEYDEF	Key definitions XAB field definitions
$XABPRODEF	Protection XAB field definitions
$XABRDTDEF	Revision date/time XAB definitions
$XABSUMDEF	Summary XAB field definitions
$XABTRMDEF	Terminal control XAB field definitions
$XADEF	DR11–W definitions for device specific characteristics
$XFDEF	DR32 device characteristic codes
$XKDEVDEF	3271 device status block
$XKSTSDEF	Definitions for 3271 line status block (returned by IO$_RDSTATS)
$XMDEF	DMC-11 device characteristic codes
$XWDEF	System definition for software DDCMP
DTK$ROUTINES	Routine definitions for DECtalk facility
LIB$ROUTINES	Routine definitions for general purpose run-time library procedures
MTH$ROUTINES	Routine definitions for mathematics run-time library procedures
NCS$ROUTINES	Routine definitions for National Character set procedure
OTS$ROUTINES	Routine definitions for language-independent support procedures
PPL$DEF	Definitions for Parallel Processing library facility
PPL$ROUTINES	Routine definitions for Parallel Processing library facility
SMG$ROUTINES	Routine definitions for Screen Management procedures
SOR$ROUTINES	Routine definitions for Sort/Merge procedures
STR$ROUTINES	Routine definitions for string manipulation procedures

Appendix C

Using System Services—Examples

This appendix contains examples that involve accessing VMS system services from VAX FORTRAN programs. The individual examples address the following operations:

1. Calling RMS Procedures (Section C.1)
2. Synchronizing Processes Using an AST Routine (Section C.2)
3. Accessing Devices Using Synchronous I/O (Section C.3)
4. Communicating with Other Processes (Section C.4)
5. Sharing Data (Section C.5)
6. Gathering and Displaying Data at Terminals (Section C.6)
7. Creating, Accessing, and Ordering Files (Section C.7)
8. Measuring and Improving Performance (Section C.8)
9. Accessing Help Libraries (Section C.9)
10. Creating and Managing Other Processes (Section C.10)

Each example includes the source program (with comments), a sample use of the program, and explanatory notes.

C.1 Calling RMS Procedures

When you explicitly call an RMS system service, the order of the arguments in the call must correspond with the order shown in the *VMS Record Management Services Manual*. You must use commas to reserve a place in the call for every argument. If you omit an argument, the procedure uses a default value of zero. For more information on calling RMS system services, see Chapter 7.

The procedure name format is SYS$procedure_name when calling an RMS routine from FORTRAN. The following example shows a call to the RMS procedure SYS$SETDDIR. This RMS procedure sets the default directory for a process.

Source Program:

```
C                                                  SETDDIR.FOR
C
C       This program calls the RMS procedure $SETDDIR to change
C       the default directory for the process.
C
        IMPLICIT INTEGER (A - Z)
        CHARACTER*17 DIR /'[EX.PROG.FOR]'/         ❶
        STAT = SYS$SETDDIR (DIR,,)                 ❷
        IF (.NOT. STAT) TYPE *, 'ERROR'
        END
```

Sample Use:

```
$ DIRECTORY                                        ❸

Directory WORK$:[EX.PROG.FOR.CALL]

BASSUM.BAS;1      BASSUM.OBJ;1      COBSUM.COM;1      DOCOMMAND.FOR;2
GETMSG.EXE;1      GETMSG.FOR;14     GETMSG.LIS;2      GETMSG.OBJ;1
MACSUM.MAR;2      SETDDIR.FOR;3     SETDDIR.LIS;1     SHOWSUM.EXE;1
SHOWSUM.FOR;6     SHOWSUM.LIS;2     SHOWSUM.OBJ;2

Total of 15 files.

$ FORTRAN SETDDIR
$ LINK SETDDIR
$ RUN SETDDIR
$DIRECTORY                                         ❹

Directory WORK$:[EX.PROG.FOR]

CALL.DIR;1      COMU.DIR;1      DEVC.DIR;1      FIL.DIR;1
HAND.DIR;1      INTR.DIR;1      LNKR.DIR;1      MNAG.DIR;1
RMS.DIR;1       SHAR.DIR;1      SYNC.DIR;1      TERM.DIR;1

Total of 12 files.
```

Notes:

❶ The default directory name is initialized into a CHARACTER variable.

❷ The call to $SETDDIR contains one argument, the directory name, which is passed by descriptor, the default argument passing mechanism for CHARACTERs. The omitted arguments are optional, but commas are necessary to reserve places in the argument list.

❸ The DIRECTORY command shows that the following directory is the default:

WORK$:[EX.V4PROG.FOR.CALL]

This directory contains the file SETDDIR.FOR.

❹ Another DIRECTORY command shows that the default directory has changed. The following directory is the new default directory:

WORK$:[EX.PROG.FOR].

C.2 Synchronizing Processes Using an AST Routine

The following example demonstrates how to request and declare an AST procedure.

Source Program:

```
C
C                                                ASTPROC.FOR
C
C     This program sets a 10-second timer that requests
C     an AST.  The main program then performs arithmetic
C     operations for the user, interrupted after 10
C     seconds by the timer AST.
C
      IMPLICIT       INTEGER*4   (A-Z)
      INCLUDE        '($SYSSRVNAM)'
      EXTERNAL       AST_PROC
      DIMENSION      BIN_DELAY(2)
      VOLATILE       BIN_DELAY                             ❶
      CHARACTER*9    DELAY          /'0 ::10.00'/
C
C      Convert delay interval to binary and set timer
C
      RESULT = SYS$BINTIM( DELAY, BIN_DELAY)
      IF (.NOT. RESULT) CALL LIB$STOP( %VAL(RESULT))
      STATUS = SYS$SETIMR( ,BIN_DELAY, AST_PROC,)          ❷
      IF (.NOT. STATUS) CALL LIB$STOP( %VAL(STATUS))
```

```
C
C      Prompt user for 2 numbers and multiply them
C
100   TYPE *, 'Enter two integers to be multiplied.
      TYPE *, '(Enter two zeros to quit)'
      ACCEPT *, NUM1, NUM2
      PRODUCT = NUM1 * NUM2
      IF (PRODUCT .EQ. 0) GO TO 200
      TYPE *, 'The product is:', PRODUCT
      GO TO 100
200   CONTINUE
      END
C
C
      SUBROUTINE       AST_PROC
C
C     This subprogram is called as an AST procedure.
C     It prints the current time at the terminal.
C
      IMPLICIT         INTEGER (A-Z)
      INCLUDE          '($BRKDEF)'
      CHARACTER*23     CUR_TIME
      CHARACTER*18     MSG     /' The time is now: '/
      CHARACTER*41     OUT_MSG
      STRUCTURE /BRKTHRU_IOSB/
          INTEGER*2    STATUS
          INTEGER*2    NUM_WRITTEN
          INTEGER*2    NUM_TIMEOUTS
          INTEGER*2    NUM_NOBROADCAST
      END STRUCTURE
      RECORD /BRKTHRU_IOSB/  IOSB
C
C
      STATUS = LIB$DATE_TIME( CUR_TIME)
      IF (.NOT. STATUS) CALL LIB$STOP( %VAL(STATUS))
      OUT_MSG = MSG // CUR_TIME
      STYPE = BRK$C_DEVICE
      STATUS = SYS$BRKTHRUW (%VAL(1), OUT_MSG, 'SYS$OUTPUT',
     1                      %VAL(STYPE), %REF(IOSB),,,,,,)
      IF (.NOT. STATUS) CALL LIB$STOP( %VAL(STATUS))
      IF (.NOT. IOSB.STATUS) CALL LIB$STOP(%VAL(IOSB.STATUS))
C
      RETURN                                          ❸
      END
```

Sample Use:

```
$ RUN ASTPROC
Enter two integers to be multiplied.
(Enter two zeros to quit)
12, 12
The product is:         144
Enter two integers to be multiplied.
(Enter two zeros to quit)
23, 45

The time is now: 14-July-1984 13:25:10.71                ❹

The product is:         1035
Enter two integers to be multiplied.
(Enter two zeros to quit)
0, 0
$
```

Notes:

❶ If any variables or arrays are used or modified by the AST routine, you should declare them as volatile in the other routines that reference them. See Section 11.3.2.2 and the description of the VOLATILE statement in the *VAX FORTRAN Language Reference Manual.*

❷ The third parameter in the $SETIMR call is the address of the entry point of the AST procedure to be executed when the timer expires. Note that the AST procedure must be declared as EXTERNAL.

❸ If you want the AST routine to be executed repeatedly, rather than just once, you can reset the timer at the end of the AST routine.

❹ When the AST is delivered, the AST routine interrupts the program and outputs a message.

C.3 Accessing Devices Using Synchronous I/O

The following example performs output to a terminal via the SYS$QIOW system service.

Source Program:

```
C                                                       QIOW.FOR
C
C       This program demonstrates the use of the $QIOW
C       system service to perform synchronous I/O to
C       a terminal.
C
        IMPLICIT          INTEGER*4 (A - Z)
        INCLUDE           '($SYSSRVNAM)'
        INCLUDE           '($IODEF)'
        CHARACTER*24      TEXT_STRING /'This is from a SYS$QIOW.'/ ❶
        CHARACTER*11      TERMINAL /'SYS$COMMAND'/
        INTEGER*2         TERM_CHAN
        STRUCTURE /TT_WRITE_IOSB/
            INTEGER*2     STATUS
            INTEGER*2     BYTES_WRITTEN
            INTEGER*4     %FILL
        END STRUCTURE
        RECORD /TT_WRITE_IOSB/  IOSB
C
C       Assign the channel number
C
        STAT = SYS$ASSIGN (TERMINAL, TERM_CHAN,,)
        IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT))                     ❷
C
C       Output the message twice
C
        DO I=1,2
           STAT = SYS$QIOW (%VAL(1),%VAL(TERM_CHAN),                   ❸
     1                      %VAL(IO$_WRITEVBLK),IOSB,,,
     1                      %REF(TEXT_STRING),
     1                      %VAL(LEN(TEXT_STRING)),,
     1                      %VAL(32),,)

        IF (.NOT. STAT) CALL LIB$STOP (%VAL(STATUS))
        IF (.NOT. IOSB.STATUS) CALL LIB$STOP (%VAL(IOSB.STATUS))
        ENDDO
        END
```

Sample Use:

```
$ FORTRAN QIOW
$ LINK QIOW
$ RUN QIOW
This is from a SYS$QIOW.
This is from a SYS$QIOW.
```

Notes:

❶ If SYS$QIO and a SYS$WAITFR are used instead of SYS$QIOW, you must use a VOLATILE declaration for any program variables and arrays that can be changed while the operation is pending.

❷ TERM_CHAN receives the channel number from the SYS$ASSIGN system service.

The process permanent logical name SYS$COMMAND is assigned to your terminal when you log in. The SYS$ASSIGN system service translates the logical name to the actual device name.

❸ SYS$QIO and SYS$QIOW accept the CHAN argument by immediate value, unlike SYS$ASSIGN, which requires that it be passed by reference. Note the use of %VAL in the call to SYS$QIOW but not in the call to SYS$ASSIGN.

The function IO$_WRITEVBLK requires values for parameters P1, P2, and P4.

- P1 is the starting address of the buffer containing the message. So, TEXT_STRING is passed by reference.
- P2 is the number of bytes to be written to the terminal. A 21 is passed, since it is the length of the message string.
- P4 is the carriage control specifier; a 32 indicates single space carriage control.

A SYS$QIOW is issued, ensuring that the output operation will be completed before the program terminates. Change the SYS$QIOW to a SYS$QIO and see what happens.

C.4 Communicating with Other Processes

The following example shows how to create a global pagefile section and how two processes can use it to access the same data. One process executes the program PAGEFIL1, which creates and writes to a global pagefile section. PAGEFIL1 then waits for a second process to update the section. The second process executes PAGEFIL2, which maps and updates the pagefile section.

Because PAGEFIL2 maps to the temporary global pagefile section created in PAGEFIL1, PAGEFIL1 must be run first. The two processes coordinate their activity through common event flags.

Source Program:

```
C                                                    PAGEFIL1.FOR
C
C     This program creates and maps a global page frame section.
C     Data in the section is accessed through an array.
C
      IMPLICIT INTEGER*4       (A-Z)
      INCLUDE                  '($SECDEF)'
      INCLUDE                  '($SYSSRVNAM)'
      DIMENSION                MY_ADR(2)
      COMMON /MYCOM/           IARRAY(50)              ❶
      CHARACTER*4              NAME/'GSEC'/
      VOLATILE /MYCOM/                                 ❷
C
C     Associate with common cluster MYCLUS
C
      CALL SYS$ASCEFC (%VAL(64),'MYCLUS',,)            ❸
C
C     Get the starting and ending addresses of the section
C
      MY_ADR(1) = %LOC(IARRAY(1))                      ❶
      MY_ADR(2) = %LOC(IARRAY(50))
      SEC_FLAGS = SEC$M_PAGFIL.OR.SEC$M_GBL.OR.SEC$M_WRT.OR.SEC$M_DZRO
C
C     Create and map the temporary global section
C
      STATUS = SYS$CRMPSC(MY_ADR,,,%VAL(SEC_FLAGS),     ❹
     1                   NAME,,,%VAL(0),%VAL(1),,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C
C     Manipulate the data in the global section         ❺
C
      DO 10 I  = 1,50
         IARRAY(I) = I
10    CONTINUE
C
      STATUS = SYS$SETEF(%VAL(72))
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      TYPE *,'Waiting for PAGEFIL2 to update section'
      STATUS = SYS$WAITFR(%VAL(73))
C
C     Print the modified pages
C
      TYPE *, 'Modified data in the global section:'
      WRITE (6,100) (IARRAY(I), I=1,50)
100   FORMAT(10I5)
      END
```

```
C                                         PAGEFIL2.FOR
C
C      This program maps and modifies a global section
C      after PAGEFIL1 creates the section.  Programs
C      PAGEFIL1 and PAGEFIL2 synchronize the processing
C      of the global section through the use of common
C      event flags.
C
      IMPLICIT INTEGER*4       (A - Z)
      INCLUDE                  '($SECDEF)'
      INCLUDE                  '($SYSSRVNAM)'
      DIMENSION                MY_ADR(2)                  ❶
      COMMON /MYCOM/           IARRAY(50)
      VOLATILE /MYCOM/                                    ❷
C
      MY_ADR(1) = %LOC(IARRAY(1))
      MY_ADR(2) = %LOC(IARRAY(50))
C
C      Associate with common cluster MYCLUS and wait for
C      event flag to be set
C
      STATUS = SYS$ASCEFC(%VAL(64),'MYCLUS',,)            ❸
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      STATUS = SYS$WAITFR (%VAL(72))
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
C
C      Set flag to allow section to be written
C
      FLAGS = SEC$M_WRT
C
C      Map the global section
C
      STATUS = SYS$MGBLSC(MY_ADR,,,%VAL(FLAGS),'GSEC',,)  ❻
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
C
C      Print out the data in the global section and        ❼
C      multiply each value by two
C
      TYPE *, 'Original data in the global section:'
      WRITE (6,100) (IARRAY(I), I=1,50)
100   FORMAT (10I5)
      DO 10 I=1,50                                        ❽
      IARRAY(I) = IARRAY(I) * 2
10    CONTINUE
C
C      Set an event flag to allow PAGEFIL1 to continue execution
C
      STATUS = SYS$SETEF(%VAL(73))
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      END
```

The options file PAGEFIL.OPT contains the following line of source text:

```
PSECT_ATTR=MYCOM,PAGE                                    ❶
COLLECT=SHARED_CLUS,MYCOM                                ❾
```

Sample Use

```
$ FORTRAN PAGEFIL1
$ FORTRAN PAGEFIL2
$ LINK PAGEFIL1,PAGEFIL/OPTIONS                          ❶
$ LINK PAGEFIL2,PAGEFIL/OPTIONS                          ❶

$ RUN PAGEFIL1                                      !****Process 1****
Waiting for PAGEFIL2 to update section
Modified data in the global section:
     2     4     6     8    10    12    14    16    18    20
    22    24    26    28    30    32    34    36    38    40
    42    44    46    48    50    52    54    56    58    60
    62    64    66    68    70    72    74    76    78    80
    82    84    86    88    90    92    94    96    98   100
$

$ RUN PAGEFIL2                                      !****Process 2****
Original data in the global section:
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50
$
```

Notes:

❶ The $CRMPSC system service maps pages starting at page boundaries. Because all named common blocks in VAX FORTRAN are longword (not page) aligned, you must ensure that IARRAY starts on a page boundary. The PSECT construct in the options file for the linker accomplishes this. PAGEFIL1 and PAGEFIL2 are linked with an options file.

❷ If any variables or arrays are used or modified by the AST routine, you should declare them as volatile in the other routines that reference them. See Section 11.3.2.2 and the description of the VOLATILE statement in the *VAX FORTRAN Language Reference Manual.*

❸ Associate to a common event flag cluster to coordinate activity. The processes must be in the same UIC group.

❹ The $CRMPSC system service creates and maps a global pagefile section.

The starting and ending process virtual addresses of the section are placed in MY_ADR. The output argument SYS_ADR receives the starting and ending system virtual addresses. The flag SEC$M_GLOBAL requests a global section. The flag SEC$M_WRT indicates that the pages should be writable as well as readable. The SEC$M_DZRO flag requests pages filled with zeros. The SEC$M_PAGFIL flag requests a temporary pagefile section.

❺ Data is written to the pagefile section.

❻ PAGEFIL2 maps the existing section as writable by specifying the SEC$M_WRT flag.

❼ Data is read from the pagefile section.

❽ Data is modified in the pagefile section.

❾ The COLLECT option instructs the linker to create a cluster named SHARED_CLUS and to put the PSECT MYCOM into that cluster. This prevents the problem of inadvertently mapping another PSECT in a page containing all or part of MYCOM. Clusters are always positioned on page boundaries.

C.5 Sharing Data

The program called SHAREDFIL is used to update records in a relative file. The SHARE qualifier is specified on the OPEN statement to invoke the RMS file sharing facility. In this example, the same program is used to access the file from two processes:

Source Program:

```
C                                           SHAREDFIL.FOR
C
C       This program can be run from two or more processes
C       to demonstrate the use of an RMS shared file to share
C       data.  The program requires the existence of a
C       relative file named REL.DAT.
C
        IMPLICIT        INTEGER*4 (A - Z)
        CHARACTER*20    RECORD
        INCLUDE         '($FORIOSDEF)'                      ❶
```

```
C
      OPEN (UNIT=1, FILE='REL', STATUS='OLD', SHARED,          ❷
     1      ORGANIZATION='RELATIVE', ACCESS='DIRECT',          ❸
     1      FORM='FORMATTED')
C
C      Request record to be examined
C
100   TYPE 10
10    FORMAT ('$Record number (CTRL Z to quit): ')
      READ (*,*, END=1000) REC_NUM
C
C      Get record from file
C
      READ (1,20, REC=REC_NUM, IOSTAT=STATUS)
     1      REC_LEN, RECORD
20    FORMAT (Q, A)

C
C              Check I/O status
C
      IF (STATUS .EQ. 0) THEN
             TYPE *, RECORD(1:REC_LEN)                         ❺
      ELSE IF (STATUS .EQ. FOR$IOS_ATTACCNON) THEN
             TYPE *,  'Nonexistent record.'
             GOTO 100
      ELSE IF (STATUS .EQ. FOR$IOS_RECNUMOUT) THEN
             TYPE *, 'Record number out of range.'
             GOTO 100
      ELSE IF (STATUS .EQ. FOR$IOS_SPERECLOC) THEN
             TYPE *, 'Record locked by someone else.'          ❹
             GOTO 100
      ELSE
             CALL ERRSNS (, RMS_STS, RMS_STV,,)
             CALL LIB$SIGNAL (%VAL(RMS_STS),
     1                        %VAL(RMS_STV))
      ENDIF
```

```
C
C     Request updated record
C
      TYPE 30
30    FORMAT ('$New Value or CR: ')
      READ (*,20) REC_LEN, RECORD
      IF (REC_LEN .NE. 0) THEN
      WRITE (1,40, REC=REC_NUM, IOSTAT=STATUS)
     1       RECORD(1:REC_LEN)
40    FORMAT (A)
      IF (STATUS .NE. 0) THEN
             CALL ERRSNS (, RMS_STS, RMS_STV,,)
             CALL LIB$SIGNAL(%VAL(RMS_STS),%VAL(RMS_STV))
          ENDIF
      ENDIF
C
C     Loop
C
      GOTO 100
C
1000  END
```

Sample Use:

```
$ FORTRAN SHAREDFIL
$ LINK SHAREDFIL
$ RUN SHAREDFIL
Record number (CTRL Z to quit): 2
MSPIGGY
New Value or CR: FOZZIE
Record number (CTRL Z to quit): 1
KERMIT
New Value or CR:
Record number (CTRL Z to quit): ^Z
$
$ RUN SHAREDFIL
Record number (CTRL Z to quit): 2    ❹
Record locked by someone else.
Record number (CTRL Z to quit): 2
Record locked by someone else.
Record number (CTRL Z to quit): 2
FOZZIE
New Value or CR: MSPIGGY
Record number (CTRL Z to quit): ^Z   ❺
$
```

Notes:

❶ The module FORIOSDEF must be included to define the symbolic status codes returned by FORTRAN I/O statements.

❷ This program requires a relative file named REL.DAT.

❸ The SHARED qualifier is used on the OPEN statement to indicate that the file can be shared. Because manual locking was not specified, RMS automatically controls access to the file. Only read and update operations are allowed in this example. No new records may be written to the file.

❹ The second process is not allowed to access record #2 while the first process is accessing it.

❺ Once the first process has finished with record #2, the second process can update it.

C.6 Displaying Data at Terminals

The following example calls SMG routines to format screen output.

No sample run is included for this example because the program requires a video terminal in order to execute properly.

Source Program:

```
C                                           SMGOUTPUT.FOR
C
C      This program calls Run-Time Library Screen Management
C      routines to format screen output.
C
      IMPLICIT INTEGER*4 (A-Z)
      INCLUDE          '($SMGDEF)'                ❶
C
C      Establish terminal screen as pasteboard
C
      STATUS = SMG$CREATE_PASTEBOARD (NEW_PID,,,) ❷
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C
C      Establish a virtual display region
C
      STATUS = SMG$CREATE_VIRTUAL_DISPLAY (15,30,DISPLAY_ID,,)       ❸
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
```

```
C
C      Paste the virtual display to the screen, starting at
C      row 5, column 15
C
      STATUS = SMG$PASTE_VIRTUAL_DISPLAY(DISPLAY_ID,NEW_PID,2,15)      ❹
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C
C      Put a border around the display area
C
      STATUS = SMG$LABEL_BORDER(DISPLAY_ID,'This is the Border',,,,,) ❺
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
C
C      Write text lines to the screen
C
      STATUS = SMG$PUT_LINE (DISPLAY_ID,' ',,,,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))                    ❻
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'Howdy, pardner',2,,,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'Double spaced lines...',2,,,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'This line is blinking',2,     ❼
     1                       SMG$M_BLINK,0,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'This line is reverse video',2,
     1                       SMG$M_REVERSE,0,,)
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      DO I = 1, 5
      STATUS = SMG$PUT_LINE (DISPLAY_ID,'Single spaced lines...',,,,,)❽
      IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))
      ENDDO
C
      END
```

Notes:

❶ The INCLUDE statement incorporates the $SMGDEF module from FORSYSDEF.TLB into the source program. This module contains symbol definitions used by the screen management routines.

❷ The call to SMG$CREATE_PASTEBOARD creates a pasteboard upon which output will be written. The pasteboard ID is returned in the variable NEW_PID.

No value is specified for the output device parameter, so the output device defaults to SYS$OUTPUT. Also, no values are specified for the PB_ROWS or PB_COLS parameters, so the pasteboard is created with the default number of rows and columns. The defaults are the number of rows and the number of columns on the physical screen of the terminal to which SYS$OUTPUT is assigned.

❸ The created virtual display is 15 lines long and 30 columns wide. The virtual display initially contains blanks.

❹ The virtual display is pasted to the pasteboard, with its upper left corner positioned at row 2, column 15 of the pasteboard. Pasting the virtual display to the pasteboard causes all data written to the virtual display to appear on the pasteboard's output device, which is SYS$OUTPUT—the terminal screen.

At this point, nothing appears on the screen because the virtual display contains only blanks. However, because the virtual display is pasted to the pasteboard, the program statements described below cause text to be written to the screen.

❺ A labeled border is written to the virtual display.

❻ Text lines are written to the virtual display. The LINE_ADV parameter specifies double spacing.

❼ These statements use the RENDITION_SET and RENDITION_COMPLEMENT parameters to display blinking and reverse video text.

❽ Single spaced text is displayed.

C.7 Creating, Accessing, and Ordering Files

In the following example, each record in a relative file is assigned to a specific cell in that file. On sequential write operations, the records are written to consecutive empty cells. Random write operations place the records into cell numbers as provided by the REC=n parameter.

Source Program:

```
C                                                 RELATIVE.FOR
C
C      This program demonstrates how to access a relative file
C      randomly. It also performs some I/O status checks.
C
      IMPLICIT          INTEGER*4 (A - Z)
      STRUCTURE /EMPLOYEE_STRUC/
         CHARACTER*5       ID_NUM
         CHARACTER*6       NAME
         CHARACTER*3       DEPT
         CHARACTER*2       SKILL
         CHARACTER*4       SALARY
      END STRUCTURE
      RECORD /EMPLOYEE_STRUC/ EMPLOYEE_REC
      INTEGER*4 REC_LEN
      INCLUDE   '($FORIOSDEF)'                          ❶
C
      OPEN (UNIT=1, FILE='REL', STATUS='OLD',           ❷
     1     ORGANIZATION='RELATIVE', ACCESS='DIRECT',
     1     FORM='UNFORMATTED',RECORDTYPE='VARIABLE')
C
C      Get records by record number until e-o-f
C              Prompt for record number
C
100   TYPE 10
10    FORMAT ('$Record number: ')
      READ (*,*, END=1000) REC_NUM                      ❸
C
C              Read record by record number
C
      READ (1,REC=REC_NUM,IOSTAT=STATUS) EMPLOYEE_REC
C
C              Check I/O status
C
      IF (STATUS .EQ. 0) THEN
             WRITE (6) EMPLOYEE_REC                     ❹
      ELSE IF (STATUS .EQ. FOR$IOS_ATTACCNON) THEN
             TYPE *,  'Nonexistent record.'
      ELSE IF (STATUS .EQ. FOR$IOS_RECNUMOUT) THEN
             TYPE *, 'Record number out of range.'
      ELSE
             CALL ERRSNS (, RMS_STS, RMS_STV,,)         ❺
             CALL LIB$SIGNAL (%VAL(RMS_STS),
     1                      %VAL(RMS_STV))
      ENDIF
C
C              Loop
C
      GOTO 100
1000  END
```

Sample Use:

```
$ FORTRAN RELATIVE
$ LINK RELATIVE
$ RUN RELATIVE
Record number: 7
08001FLANJE119PL1920
Record number: 1
07672ALBEHA210SE2100
Record number: 30
Nonexistent record.
Record number: ^Z
$
```

Notes:

❶ The INCLUDE statement defines all FORTRAN I/O status codes.

❷ The OPEN statement defines the file and record processing characteristics. Although the file organization is specified as relative, RMS would in fact obtain the file organization from an existing file. If the file's organization were not relative, the file OPEN statement would fail.

The file is being opened for unformatted I/O because the data records will be read into a VAX FORTRAN record (EMPLOYEE_REC), and VAX FORTRAN does not allow records to be used in formatted I/O.

❸ The READ statement reads the record specified in REC_NUM, rather than the next consecutive record. The status code for the record operation is returned in the variable STATUS.

❹ These statements test the record operation status obtained in comment 3. Note, the status codes returned by RMS and VAX FORTRAN are not numerically or functionally similar.

❺ RMS status codes actually require two parameters. These values can be obtained using the ERRSNS subroutine.

C.8 Measuring and Improving Performance

This example demonstrates how to adjust the size of the process working set from a program.

Source Program:

```
C                                                 ADJUST.FOR
C
C      This program demonstrates how a program can control
C      its working set size using the $ADJWSL system service.
C
      IMPLICIT      INTEGER (A-Z)
      INCLUDE       '($SYSSRVNAM)'
      INTEGER*4     ADJUST_AMT      /0/
      INTEGER*4     NEW_LIMIT       /0/
C
      CALL LIB$INIT_TIMER
C
      DO 100 ADJUST_AMT= -50,70,10
C
C      Modify working set limit
C
      RESULT = SYS$ADJWSL( %VAL(ADJUST_AMT), NEW_LIMIT)    ❶
      IF (.NOT. RESULT) CALL LIB$STOP(%VAL(RESULT))
C
      TYPE 50, ADJUST_AMT, NEW_LIMIT
50    FORMAT(' Modify working set by', I4,
     1       '   New working set size =', I4)
100   CONTINUE
C
      CALL LIB$SHOW_TIMER
      END
```

Sample Use:

```
$ SET WORKING_SET/NOADJUST                                      ❷
$ SHOW WORKING_SET
  Working Set       /Limit=  150    /Quota=  200          /Extent=  200
  Adjustment disabled     Authorized Quota=  200 Authorized Extent=  200
$ FORTRAN ADJUST
$ LINK ADJUST
$ RUN ADJUST
Modify working set by -50    New working set size = 100
Modify working set by -40    New working set size = 60
Modify working set by -30    New working set size = 41
Modify working set by -20    New working set size = 41          ❸
Modify working set by -10    New working set size = 41
Modify working set by   0    New working set size = 41
Modify working set by  10    New working set size = 51
Modify working set by  20    New working set size = 71
Modify working set by  30    New working set size = 101
Modify working set by  40    New working set size = 141
Modify working set by  50    New working set size = 191
Modify working set by  60    New working set size = 200         ❹
Modify working set by  70    New working set size = 200
 ELAPSED:    0 00:00:01.58  CPU: 0:00:00.11  BUFIO: 1  DIRIO: 2  FAULTS: 25

$
```

Notes:

❶ The SYS$ADJWSL is used to increase or decrease the number of pages in the process working set.

❷ The DCL SHOW WORKING_SET command displays the current working set limit and the maximum quota.

❸ Notice that the program cannot decrease the working set limit beneath the minimum established by the operating system.

❹ Similarly, the process working set cannot be expanded beyond the authorized quota.

C.9 Accessing Help Libraries

The following example demonstrates how to obtain text from a help library. After the initial help request has been satisfied, the user is prompted and can request additional information.

Source Program:

```
C                                                HELPOUT.FOR
C
C      This program satisfies an initial help request
C      and enters interactive HELP mode.  The library
C      used is SYS$HELP:HELPLIB.HLB.
C
      IMPLICIT       INTEGER*4 (A - Z)
      CHARACTER*32   KEY
      EXTERNAL       LIB$PUT_OUTPUT,LIB$GET_INPUT     ❶
C
C      Request a HELP key
C
      WRITE (6,2000)
2000  FORMAT(1X,'What Topic would you like HELP with? ',$)
      READ (5,1000) KEY
1000  FORMAT (A32)
C
C      Locate and print the help text
C
      STATUS = LBR$OUTPUT_HELP(LIB$PUT_OUTPUT,,KEY,   ❷
     1                        'HELPLIB',,LIB$GET_INPUT)
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      END
```

Sample Use:

```
$ FORTRAN HELPOUT
$ LINK HELPOUT
$ RUN HELPOUT
What topic would you like HELP with? TYPE

TYPE
    Displays the contents of a file or a group of files on the
    current output device.

    Format:

        TYPE file-spec[,...]

    Additional information available:

    Parameters  Qualifiers
    /OUTPUT

TYPE Subtopic? /OUTPUT

TYPE
     /OUTPUT

          /OUTPUT=file-spec

     Requests that the output from the TYPE command be written
     to the specific file, rather than to SYS$OUTPUT.

TYPE Subtopic? ^Z
$
```

Notes:

❶ To pass the address of LIB$PUT_OUTPUT and LIB$GET_INPUT, they must be declared as EXTERNAL. You can supply your own routines for handling input and output.

❷ The address of an output routine is a required argument. When requesting prompting mode, the default mode, an input routine must be specified.

C.10 Creating and Managing Other Processes

The following example demonstrates how a created process can use the SYS$GETJPIW system service to obtain the PID of its creator process. It also shows how to set up an item list to translate a logical name recursively.

Source Program:

```
C                                                   GETJPI.FOR
C      This program demonstrates process creation and
C      control. It creates a subprocess then hibernates
C      until the subprocess wakes it.
C
C
      IMPLICIT       INTEGER*4 (A - Z)
      INCLUDE        '($SSDEF)'
      INCLUDE        '($LNMDEF)'
      INCLUDE        '($SYSSRVNAM)'
      CHARACTER*255   TERMINAL       /'SYS$OUTPUT'/
      CHARACTER*9     FILE_NAME      /'GETJPISUB'/
      CHARACTER*5     SUB_NAME       /'OSCAR'/
      INTEGER*4       PROCESS_ID     /0/
      CHARACTER*17    TABNAM         /'LNM$PROCESS_TABLE'/
      CHARACTER*255   RET_STRING
      CHARACTER*2     ESC_NULL
      INTEGER*4       RET_ATTRIB
      INTEGER*4       RET_LENGTH      /10/
      STRUCTURE /ITMLST3_3ITEMS/
          STRUCTURE    ITEM(3)
              INTEGER*2    BUFFER_LENGTH
              INTEGER*2    CODE
              INTEGER*4    BUFFER_ADDRESS
              INTEGER*4    RETLEN_ADDRESS
          END STRUCTURE
          INTEGER*4    END_OF_LIST
      END STRUCTURE
      RECORD /ITMLST3_3ITEMS/  TRNLST
```

```
C
C      Translate SYS$OUTPUT
C      Set up TRNLST, the item list for $TRNLNM
C
      TRNLST.ITEM(1).CODE = LNM$_STRING
      TRNLST.ITEM(1).BUFFER_LENGTH = 255
      TRNLST.ITEM(1).BUFFER_ADDRESS = %LOC(RET_STRING)
      TRNLST.ITEM(1).RETLEN_ADDRESS = 0

      TRNLST.ITEM(2).CODE = LNM$_ATTRIBUTES
      TRNLST.ITEM(2).BUFFER_LENGTH = 4
      TRNLST.ITEM(2).BUFFER_ADDRESS = %LOC(RET_ATTRIB)
      TRNLST.ITEM(2).RETLEN_ADDRESS = 0

      TRNLST.ITEM(3).CODE = LNM$_LENGTH
      TRNLST.ITEM(3).BUFFER_LENGTH = 4
      TRNLST.ITEM(3).BUFFER_ADDRESS = %LOC(RET_LENGTH)
      TRNLST.ITEM(3).RETLEN_ADDRESS = 0

      TRNLST.END_OF_LIST = 0
C
C      Translate SYS$OUTPUT
C
100   STATUS = SYS$TRNLNM (,TABNAM,TERMINAL(1:RET_LENGTH),,TRNLST)
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      IF (IAND(LNM$M_TERMINAL, RET_ATTRIB).EQ. 0) THEN
             TERMINAL = RET_STRING(1:RET_LENGTH)
             GO TO 100
      ENDIF
C
C      Check if process permanent file
C
      IF (RET_STRING(1:2) .EQ. ESC_NULL) THEN
             RET_STRING = RET_STRING(5:RET_LENGTH)
             RET_LENGTH = RET_LENGTH - 4
      ENDIF
C
C      Create the subprocess
C
      STATUS = SYS$CREPRC (PROCESS_ID, FILE_NAME,,          ❶
     1                     RET_STRING(1:RET_LENGTH),,,,
     1                     SUB_NAME,%VAL(4),,,)
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      TYPE 10, PROCESS_ID
10    FORMAT (' PID of subprocess OSCAR is ', Z)
C
C      Wait for wakeup by subprocess
C
      STATUS = SYS$HIBER ()                                 ❷
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
C
      TYPE *, 'GETJPI has been awakened.'
      END
```

```
C                                        GETJPISUB.FOR
C     This program is run in the subprocess OSCAR
C     which is created by GETJPI.  It obtains its
C     creator's PID and then wakes it.
C
      IMPLICIT       INTEGER*4 (A - Z)                     ❸
      INCLUDE        '($JPIDEF)'
      INCLUDE        '($SYSSRVNAM)'
      STRUCTURE /GETJPI_IOSB/
          INTEGER*4  STATUS
          INTEGER*4  %FILL
      END STRUCTURE
      RECORD /GETJPI_IOSB/  IOSB
      STRUCTURE /ITMLST3_1ITEM/
          STRUCTURE    ITEM
              INTEGER*2    BUFFER_LENGTH
              INTEGER*2    CODE
              INTEGER*4    BUFFER_ADDRESS
              INTEGER*4    RETLEN_ADDRESS
          END STRUCTURE
          INTEGER*4    END_OF_LIST
      END STRUCTURE
      RECORD /ITMLST3_1ITEM/  JPI_LIST
C
C     Set up buffer address for GETJPI
C
      JPI_LIST.ITEM.CODE = JPI$_OWNER                      ❹
      JPI_LIST.ITEM.BUFFER_LENGTH = 4
      JPI_LIST.ITEM.BUFFER_ADDRESS = %LOC(OWNER_PID)
      JPI_LIST.ITEM.RETLEN_ADDRESS = 0
C
C     Get PID of creator
C
      STATUS = SYS$GETJPIW (%VAL(1),,, JPI_LIST,IOSB,,)    ❺
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
      IF (.NOT. IOSB.STATUS) CALL LIB$STOP (%VAL(IOSB.STATUS))
C
C     Wake creator
C
      TYPE *, 'OSCAR is waking creator.'
      STATUS = SYS$WAKE (OWNER_PID,)
      IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
C
      END
```

Sample Use:

```
$ FORTRAN GETJPI,GETJPISUB
$ LINK GETJPI
$ LINK GETJPISUB
$ RUN GETJPI
PID of subprocess OSCAR is 530048
OSCAR is waking creator.
GETJPI has been awakened.
```

Notes:

❶ The subprocess is created using SYS$CREPRC.

❷ The process hibernates.

❸ The INCLUDE statement defines the value of all JPI$ codes including JPI$_OWNER. JPI$_OWNER is the item code which requests the PID of the owner process. If there is no owner process (that is, if the process about which information is requested is a detached process), the system service $GETJPIW returns a PID of zero.

❹ Because of the item code JPI$_OWNER in the item list, $GETJPIW returns the PID of the owner of the process about which information is requested. If the item code were JPI$_PID, $GETJPIW would return the PID of the process about which information is requested.

Because the default value of 0 is used for arguments PIDADR and PRCNAM, the process about which information is requested is the requesting process, namely, OSCAR.

❺ The item list for SYS$GETJPIW consists of a single item descriptor followed by a zero longword.

Appendix D

Compatibility: VAX FORTRAN and FORTRAN-66

VAX FORTRAN is based on American National Standard FORTRAN–77, X3.9-1978. As a result, it contains certain incompatibilities with FORTRAN implementations that are based on the previous standard, X3.9-1966. The following areas are affected:

- The minimum iteration count of DO loops
- The EXTERNAL statement
- The defaults for the OPEN statement's BLANK and STATUS keywords
- The X format edit descriptor
- The effect of attempting to open a connected unit

The VAX FORTRAN compiler selects FORTRAN–77 language interpretations by default. If you are compiling FORTRAN-66 programs, there are several actions that you can take to compensate for language incompatibilities:

- You can modify your FORTRAN-66 programs so that they are compatible with FORTRAN–77 language interpretations.

  Compiler diagnostics help you to identify OPEN statements in which you should add an explicit STATUS keyword. Linker diagnostics help you to locate EXTERNAL statements that must be changed to INTRINSIC statements.

- You can select FORTRAN-66 language interpretations by specifying the /NOF77 qualifier on your FORTRAN command line or the /NOF77 option on an OPTIONS statement. The /NOF77 option affects the interpretation of the minimum iteration counts of DO loops, the EXTERNAL statements, and the OPEN statement default for BLANK and STATUS. It does not affect the X format edit descriptor.

  You can redefine the FORTRAN command to include the /NOF77 command qualifier, thereby selecting FORTRAN-66 language interpretations by default. To redefine the FORTRAN command, use a VMS command language symbol definition of the form:

  ```
  $  FOR*TRAN :== "FORTRAN/NOF77"
  ```

  You can include this symbol definition in your LOGIN command file or in a system-wide LOGIN command file. In the latter case, the FORTRAN command is redefined for all users. The asterisk (*) in the symbol definition permits you to abbreviate the FORTRAN command to three or more characters (FOR, FORT, and so on).

This appendix discusses each of the language differences. When possible, it gives an example of how you can modify your FORTRAN-66 programs to make them compatible with both VAX FORTRAN (FORTRAN–77) and FORTRAN-66.

D.1 Minimum Iteration Count for DO Loops

In FORTRAN–77, the body of a DO loop is not executed if the end condition of the loop is already satisfied when the DO statement is executed (see Section 10.3.2). In most implementations of FORTRAN-66, however, the body of a DO loop is always executed at least once.

The /[NO]F77 compilation qualifier and the /[NO]F77 option on the OPTIONS statement both control the interpretation of the minimum iteration count of DO loops.

If you intend to have either the /F77 qualifier or the /F77 option in effect for a FORTRAN-66 program, you may want to ensure a minimum loop count of one by modifying the program's DO statements. As an example, consider the following statement in a FORTRAN-66 program:

```
DO 10, J=ISTART,IEND
```

This DO statement specifies that the body of the loop is executed only when IEND is greater than or equal to ISTART. However, you could modify the statement to handle a situation in which IEND might be less than ISTART. For example:

```
DO 10 J=ISTART,MAX(ISTART,IEND)
```

The body of this modified DO loop is executed at least once in both FORTRAN–77 and FORTRAN-66.

D.2 EXTERNAL Statement

In FORTRAN-66, the EXTERNAL statement is used to specify that a symbolic name is the name of either a user-defined external procedure or a FORTRAN-supplied function, like SQRT or SIN. An EXTERNAL statement is required to pass a procedure name as an actual argument.

In FORTRAN–77, the EXTERNAL and INTRINSIC statements accomplish this function:

- Use the INTRINSIC statement for FORTRAN-supplied intrinsic procedures (for example, SQRT).
- Use the EXTERNAL statement for user-supplied procedures.

In FORTRAN-66, EXTERNAL SQRT specifies the FORTRAN-supplied real square root function. In FORTRAN–77, the identical syntax specifies a user-defined function, and an error results at link time if there is no user-defined function called SQRT.

The /[NO]F77 compilation qualifier and the /[NO]F77 option on the OPTIONS statement both control the interpretation of the EXTERNAL statement.

When you compile a program with the /NOF77 qualifier or use the /NOF77 option in a program, EXTERNAL *SQRT specifies a user-supplied function with the same name as a FORTRAN-supplied function. This syntax is invalid in FORTRAN–77; the FORTRAN–77 EXTERNAL statement must be used instead.

You cannot modify the EXTERNAL statements in your programs so that the same source program works with both FORTRAN–77 and FORTRAN-66 in all cases; you must substitute an equivalent statement:

FORTRAN-66	FORTRAN-77
EXTERNAL USER	EXTERNAL USER (no change required)
EXTERNAL SQRT	INTRINSIC SQRT
EXTERNAL *SQRT	EXTERNAL SQRT

D.3 OPEN Statement Keyword Defaults

The FORTRAN-66 language did not contain an OPEN statement; however, many implementations based on FORTRAN-66 do contain an OPEN statement. The defaults for FORTRAN–77 OPEN statement keywords differ from traditional FORTRAN defaults for the attributes that these keywords control.

D.3.1 BLANK Keyword Default

In FORTRAN–77, the OPEN statement's BLANK keyword controls the interpretation of blanks in numeric input fields. The FORTRAN–77 default is BLANK='NULL'; that is, blanks in numeric input fields are ignored. The FORTRAN-66 interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

If a logical unit is opened without an explicit OPEN statement, VAX FORTRAN and FORTRAN-66 both provide a default equivalent to BLANK='ZERO'.

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D, E, F, G, I, O, and Z field descriptors. If BLANK='NULL' is in effect, embedded and trailing blanks are ignored; the value is converted as if the nonblank characters were right-justified in the field. If BLANK='ZERO' is in effect, embedded and trailing blanks are treated as zeros. The following example illustrates the difference in how blanks in numeric input fields are interpreted in FORTRAN–77 and in FORTRAN-66:

Source Program:

```
      OPEN (UNIT=1, STATUS='OLD')
      READ (1,10) I, J
10    FORMAT (2I5)
      END
```

Data record:

```
Δ1Δ2ΔΔΔΔ12
```

Results of READ:

FORTRAN-66	FORTRAN-77
I = 1020	I = 12
J = 12	J = 12

The /[NO]F77 compilation qualifier and the /[NO]F77 option both control the default value for the BLANK keyword. If your program treats blanks in numeric input fields as zeros and you do not want to use either the /NOF77 qualifier or the /NOF77 option, either include BLANK='ZERO' in the OPEN statement or use the BZ edit descriptor in the FORMAT statement.

D.3.2 OPEN Statement's STATUS Keyword Default

In FORTRAN–77, the OPEN statement's STATUS keyword specifies the initial status of the file ('OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'). The FORTRAN–77 default is STATUS='UNKNOWN'; that is, an existing file is opened or a new file is created if the file does not exist. DIGITAL's implementation of FORTRAN based on FORTRAN-66 had a keyword TYPE, which is a synonym for STATUS; however, the default for TYPE is TYPE='NEW'.

The /[NO]F77 compilation qualifier and the /[NO]F77 option on the OPTIONS statement both control the default value for the STATUS (or TYPE) keyword.

If you use either the /F77 compilation qualifier or the /F77 option and you do not specify STATUS (or TYPE) in an OPEN statement, the compiler issues an informational message to warn you that it is using a default of STATUS='UNKNOWN'. It is advisable to include an explicit STATUS (or TYPE) keyword in every OPEN statement. If you do not, files that you wish to retain may be overwritten. For example, a program that is expected to write a new file will overwrite any existing version of that file if the file's status is unknown; in some cases, this may not be the result that you want.

D.4 X Format Edit Descriptor

The nX edit descriptor causes transmission of the next character to or from a record to occur at the position n characters to the right of the current position. In a FORTRAN-77 output statement, character positions that are skipped are not modified, and the length of the output record is not affected. However, in many FORTRAN-66 implementations, the X edit descriptor writes blanks and may extend the output record. For example, consider the following statements:

```
     WRITE (1,10)
10   FORMAT (1X,'ABCDEF',T4,2X,'12345',3X)
```

These statements produce the following output records:

FORTRAN-66	FORTRAN-77
ΔABΔΔ12345ΔΔΔ	ΔABCD12345

Neither the /[NO]F77 compilation qualifier nor the /[NO]F77 option on the OPTIONS statement affect the interpretation of the X edit descriptor. To achieve the FORTRAN-66 effect, you must change nX to n(' ').

D.5 Open Operation on a Connected Unit

In FORTRAN-66, it is an error to execute an OPEN statement on a logical unit that already has a file connected to it.

In FORTRAN–77, the behavior varies as follows:

- If the file specification (or the default) matches that of the currently opened file, the new value (if any) of the BLANK keyword is used and the new open request is otherwise ignored.
- If the file specifications do not match, the currently open file is closed and the new file is opened.

In neither case is an error issued for FORTRAN–77.

There is no way to cause programs compiled with VAX FORTRAN to exhibit the FORTRAN-66 behavior.

Appendix E

Compatibility: VAX FORTRAN and PDP-11 FORTRAN

VAX FORTRAN is a compatible superset of PDP-11 FORTRAN IV and PDP-11 FORTRAN-77. Most PDP-11 FORTRAN programs can run on VAX systems without modifications. Execution may be affected in some cases, however, due to differences between the hardware architecture of PDP-11 and VAX computers and differences between the IAS/RSX-11 and VMS operating environments. Execution also may be affected by the FORTRAN-77 language interpretations described in Appendix D.

The issues discussed in this section concern differences in language, run-time support, and utilities provided in the form of subroutines.

E.1 Language Differences

Differences related to language involve the following areas:

- Logical tests
- Floating-point results
- Character and Hollerith constants
- Logical unit numbers
- Assigned GO TO label list
- Effect of DISPOSE='PRINT' specification

E.1.1 Logical Tests

The logical constants .TRUE. and .FALSE. are defined, respectively, as all ones and all zeros by both VAX FORTRAN and PDP–11 FORTRAN. The test for .TRUE. and .FALSE. differs, however.

- VAX FORTRAN tests the low-order bit (bit 0) of a logical value. This is the system-wide VAX convention for testing logical values.
- PDP–11 FORTRAN–77 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 zeros is a .FALSE. value, and any nonzero bit pattern is a .TRUE. value.

In most cases, this difference has no effect. It is significant only for nonstandard FORTRAN programs that perform arithmetic operations on logical values and then make logical tests on the result. For example:

```
LOGICAL*1 BA
BA = 3
IF (BA) GO TO 10
```

VAX FORTRAN produces a value of .TRUE., PDP–11 FORTRAN–77 produces .FALSE., and PDP–11 FORTRAN IV produces .TRUE.

E.1.2 Floating-Point Results

Differences in results from math library routines may occur because of new implementations of these routines that take advantage of the VAX instruction set. The VAX functions produce results with an accuracy equal to or greater than the corresponding PDP–11 functions, but there may be differences.

In addition, floating-point constants without exponents are not immediately converted to REAL*4, as is the case with PDP–11 FORTRAN. This feature provides greater accuracy when such constants are used in double-precision expressions.

E.1.3 Character and Hollerith Constants

VAX FORTRAN supports both Hollerith constants, with the notation nHa...a, and character constants, with the notation 'a...a'. In PDP–11 FORTRAN-IV, both notations are used for Hollerith constants. (Note that Hollerith constants have no data type; Hollerith constants assume a data type consistent with their use.)

In most cases, the conflicting use of the 'a...a' notation is not a problem; VAX FORTRAN 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 'a...a' notation is always passed as a character constant, never as Hollerith. For example, consider the following source code:

```
SUBROUTINE S(F)
     .
     .
     .
CALL F('ABCD')
```

If the subroutine referenced by F expects a Hollerith constant (that is, the dummy argument is a numeric data type), execution is not correct. The actual and dummy arguments must agree in data type. This is not the case in the preceding example. To avoid this problem, you must change to the nHa...a notation, as follows:

```
SUBROUTINE S(F)
     .
     .
     .
CALL F(4HABCD)
```

E.1.4 Logical Unit Numbers

If you do not specify a logical unit number in an I/O statement, a default unit number is used. The defaults used by VAX FORTRAN differ from those used by PDP-11 FORTRAN-77, as shown in Table E-1.

Table E-1: Default Logical Unit Numbers

I/O Statement	PDP-11 Unit	VAX Unit
READ	1	-4
PRINT	6	-1
TYPE	5	-2
ACCEPT	5	-3

Note that PDP-11 FORTRAN-77 uses normal logical unit numbers; VAX FORTRAN uses unit numbers that are unavailable to users. This feature prevents conflicts between I/O statements that use the default logical unit numbers and those that use explicit logical unit numbers. This should have no visible effect on program execution.

E.1.5 Assigned GO TO Label List

The labels specified in an assigned GO TO label list are checked by the VAX FORTRAN compiler to ensure their validity in the program unit. However, VAX FORTRAN, like PDP-11 FORTRAN IV, does not perform a check at run time to ensure that a label actually assigned is in the list. PDP-11 FORTRAN-77 does perform this check at run time.

E.1.6 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 FORTRAN, the file is retained after being printed.

E.2 Run-Time Support Differences

Differences in run-time support between VAX FORTRAN and PDP-11 FORTRAN are reflected in run-time error numbers, in run-time error reporting, and in some values for OPEN statement keywords.

E.2.1 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 were either deleted from or redefined in the VAX Run-Time Library. The following error numbers are affected.

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 GO TO OUT OF RANGE)	Deleted; no error is generated by the VAX hardware when this condition occurs. Program execution continues in line.
92 (ASSIGNED LABEL NOT IN LIST)	Deleted; as described in Section E.1.5, VAX FORTRAN 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 FORTRAN.

E.2.2 Error Handling and Reporting

VAX FORTRAN differs from PDP-11 FORTRAN-77 in the way it treats error continuation, I/O errors, and OPEN/CLOSE statement errors. Section 5.1 describes Run-Time Library error handling.

E.2.2.1 Continuing After Errors

In PDP-11 FORTRAN, program execution after errors, such as floating overflows, normally continues until 15 such errors occur. At that point, execution is terminated. VAX FORTRAN, however, sets a limit of one such error; program execution normally terminates when the first 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 9.
- Include the ERRSET subroutine (see Section E.3.3). ERRSET alters the Run-Time Library's default error processing to match the behavior of PDP-11 FORTRAN-77.

E.2.2.2 I/O Errors with IOSTAT or ERR Specified

If an IOSTAT or ERR specification was included in the I/O statement, VAX FORTRAN neither generates an error message nor increments the image error count when an I/O error occurs. Under these circumstances, PDP-11 FORTRAN both reports the error and increments the task error count.

E.2.2.3 OPEN/CLOSE Statement Errors

Unlike PDP-11 FORTRAN, VAX FORTRAN reports only the first error encountered in an OPEN or CLOSE statement. PDP-11 FORTRAN reports all errors detected in processing these statements.

E.2.3 OPEN Statement Keywords

For VAX FORTRAN, the space requested by the INITIALSIZE keyword is allocated contiguously, if possible, on what is called a best-try basis. That is, if you specify an INITIALSIZE value and sufficient contiguous space is available, allocation is contiguous. If not enough contiguous space is available, allocation is noncontiguous.

For PDP–11 FORTRAN–77, allocation of contiguous or noncontiguous space depends on the sign of the value specified for the INITIALSIZE and EXTENDSIZE keywords. To be compatible with PDP–11 FORTRAN, VAX FORTRAN uses the absolute value of the user-supplied value.

E.3 Utility Subroutines

A number of utility subroutines are available for use with PDP–11 FORTRAN–77. All are supplied as part of PDP–11 FORTRAN–77, as described in the *PDP–11 FORTRAN–77 User's Guide*.

The following subroutines are supplied as a standard part of VAX FORTRAN:

DATE
ERRSNS
EXIT
IDATE
SECNDS
TIME

See the discussion of system subroutines in the *VAX FORTRAN Language Reference Manual* for more information about these subroutines.

The remaining subroutines are provided only for purposes of compatibility; most have been superseded by features included in VAX FORTRAN, while others have little applicability on VMS systems. The following subroutines fall into this category:

ASSIGN
CLOSE
ERRSET
ERRTST
FDBSET
IRAD50
RAD50

RAN
RANDU
R50ASC
USEREX

Sections E.3.1 through E.3.11 describe these routines.

E.3.1 ASSIGN Subroutine

The ASSIGN subroutine enables you to assign a device or file 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 or INQUIRE 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 FILE keyword in an OPEN statement or use the ASSIGN system command.

A call to the ASSIGN subroutine has the form:

```
CALL ASSIGN (n[,name][,icnt])
```

n
Is an integer value specifying the logical unit number.

name
Is a variable, array, array element, or character constant containing any standard file specification.

icnt
Is an INTEGER*2 value that specifies the number of characters contained in the string name.

Notes

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, the name argument must also be specified.

E.3.2 CLOSE Subroutine

The CLOSE subroutine closes the file currently open on a logical unit. A call to the CLOSE subroutine has the form:

```
CALL CLOSE (n)
```

n
Is an integer value specifying the logical unit.

Notes

After the file is closed, the logical unit again assumes the default file-name specification.

E.3.3 ERRSET Subroutine

The ERRSET subroutine determines the action taken in response to an error detected by the Run-Time Library. The VMS condition handling facility provides a more general method of defining actions to be taken when errors are detected (see Chapter 9). A call to the ERRSET subroutine has the form:

```
CALL ERRSET (number, contin, count, type, log, maxlim)
```

number
Is an integer value specifying the error number.

contin
Is a logical value:

.TRUE.	Continue after error is detected.
.FALSE.	Exit after error is detected.

count
Is a logical value:

.TRUE.	Count the error against the maximum error limit.
.FALSE.	Do not count the error against the maximum error limit.

type
Is a logical value:

.TRUE.	Pass control to an ERR transfer label, if specified.
.FALSE.	Return to routine that detected the error, for default error recovery.

log
Is a logical value:

.TRUE.	Produce an error message for this error.
.FALSE.	Do not produce an error message for this error.

maxlim
Is a positive INTEGER*2 value specifying the maximum error limit. The default is set to 15 at program initialization.

Notes

- The error action specified for each error is independent of other errors.
- Null arguments are legal for all arguments, except number arguments, and have no effect on the current state of that argument.
- An external reference to the ERRSET subroutines 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.

E.3.4 ERRTST Subroutine

The ERRTST subroutine checks for a specific error and resets the error flag for that error. To perform appropriate actions in response to errors, you should establish a condition handler, as described in Chapter 9. A call to the ERRTST subroutine has the form:

```
CALL ERRTST (i,j)
```

i
Is an integer value specifying the error number.

j
Is a variable used for return value of error check:

j = 1:	Error i has occurred.
j = 2:	Error i has not occurred.

Notes

- The ERRTST subroutine is independent of the ERRSET subroutine; neither subroutine has any direct effect on the other.
- An external reference to the ERRTST subroutines 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.

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).

E.3.5 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. A call to the FDBSET subroutine has the form:

```
CALL FDBSET (unit[,acc,share,numbuf,initsz,extend])
```

unit
Is an integer value specifying the logical unit.

acc
Is a character constant specifying the access mode to be used:

'READONLY'	Establish 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
Is a character constant 'SHARE' indicating that shared access is allowed.

numbuf
Is an INTEGER*2 value specifying the number of buffers to be used for multibuffered I/O.

initsz
Is an INTEGER*2 value specifying the number of blocks initially allocated for a new file.

extend
Is an INTEGER*2 value specifying the number of blocks by which to extend a file.

Notes

- FDBSET can be used only before issuing the first I/O statement for the unit.
- The FDBSET and ASSIGN subroutines and the DEFINE FILE statement can be used together, but none can be used in conjunction with the OPEN statement or INQUIRE statement for the same unit.
- The unit argument must be specified. All other arguments are optional.

E.3.6 IRAD50 Subroutine

The IRAD50 subroutine is used to convert Hollerith data to Radix-50 form. The IRAD50 subroutine can be called as a function subprogram if the return value is desired (format 1), or as a subroutine if the return value is not desired (format 2):

- Format 1: `n = IRAD50 (icnt,input,output)`
- Format 2: `CALL IRAD50 (icnt,input,output)`

n
Is an INTEGER*2 value indicating how many characters are converted.

icnt
Is an INTEGER*2 value specifying the maximum number of characters to be converted.

input
Is a Hollerith string to be converted to Radix–50.

output
Is a numeric variable or array element where the Radix–50 results are stored.

Notes

- 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.
- Scanning of the input characters terminates on the first non-Radix–50 character in the input string.

E.3.7 RAD50 Function

The RAD50 function subprogram provides a simplified way to encode six Hollerith characters as two words of Radix–50 data. It has the form:

```
RAD50 (name)
```

name
Is a numeric variable name or array element corresponding to a Hollerith string.

Notes

The RAD50 function is equivalent to the following source statements:

```
FUNCTION RAD50 (A)
CALL IRAD50 (6,A,RAD50)
RETURN
END
```

E.3.8 RAN Function

The RAN function subprogram returns a pseudo-random number as the function value. It has the form:

```
RAN (i1,i2)
```

i1,i2
Are INTEGER*2 variables or array elements that contain the seed for computing the random number. The values of i1 and i2 are updated during the computation to contain the updated seed.

Notes

- The algorithm for computing the random number value is identical to the algorithm used in the RANDU subroutine (see Section E.3.9).
- The RAN function is equivalent to the following source statements:

```
FUNCTION RAN (I1,I2)
CALL RANDU (I1,I2,RAN)
RETURN
END
```

- This RAN function is distinguished from the single argument RAN function by the number of arguments. The single argument form uses a statistically better algorithm and is recommended when compatibility with PDP–11 FORTRAN is not important.

E.3.9 RANDU Subroutine

The RANDU subroutine computes a pseudo-random number as a single-precision value uniformly distributed in the following range:

```
0.0 .LE. value .LT. 1.0
```

A call to the RANDU subroutine has the form:

```
CALL RANDU (i1,i2,x)
```

i1,i2
Are INTEGER*2 variables or array elements that contain the seed for computing the random number. The values of i1 and i2 are updated during the computation to contain the updated seed.

x
Is a real variable or array element where the computed random number is stored.

Notes

The algorithm for computing the random number value is as follows:

If I1=0, I2=0, set the generator base as follows:

```
X(n+1) = 2**16 + 3
```

Otherwise, set it as follows:

```
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.

E.3.10 R50ASC Subroutine

The R50ASC subroutine converts Radix–50 values to Hollerith strings. A call to the R50ASC subroutine has the form:

```
   CALL R50ASC (icnt,input,output)
```

icnt
Is an INTEGER*2 value specifying the number of ASCII characters to be produced.

input
Is a numeric variable or array element containing the Radix–50 data. The number of words of input equals (icnt+2)/3.

output
Is a numeric variable or array element where the Hollerith characters are to be stored.

Notes

If the undefined Radix-50 code is detected or if the Radix-50 word exceeds 174777 (octal), question marks are placed in the output location.

E.3.11 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. A call to the USEREX subroutine has the form:

```
CALL USEREX (name)
```

name
Specifies the name of the routine to be called. The routine name must appear in an EXTERNAL statement in the program unit.

Notes

- The user exit subroutine is called as a VMS termination handler. See the *VMS System Services Reference Manual* for information regarding termination handlers.
- Do not attempt to perform FORTRAN I/O operations as part of an exit handler.

Appendix F

Diagnostic Messages

Diagnostic messages related to a VAX FORTRAN program can come from the compiler, the linker, or the VAX run-time system:

- The VAX FORTRAN compiler detects syntax errors in the source program, such as unmatched parentheses, invalid characters, misspelled keywords, and missing or invalid parameters.
- The VAX linker detects errors in object file format and source program errors such as undefined symbols.
- The VAX run-time system reports errors that occur during execution.

These messages are displayed on your terminal or in your log file. The format of the messages is as follows:

```
%SOURCE-CLASS-MNEMONIC, message_text
```

The contents of the fields of information in diagnostic messages are as follows:

%	The percent sign identifies the line as a message.
SOURCE	A two-, three-, or four-letter code that identifies the origin of the message; that is, whether it came from the compiler (FORT), the linker (LINK), or the run-time system (FOR, SS, or MTH).
CLASS	A single character that determines message severity. The four classes of error messages are: Fatal (F), Error (E), Warning (W), and Informational (I). The definition of each class depends on the source of the message. Definitions for each of the classes are given in the section that details the error messages given by a particular source.
MNEMONIC	A 6- to 9-character name that is unique to that message.
message_text	Explains the event that caused the message to be generated.

This appendix lists and describes the messages issued by the compiler and the run-time system. It also provides a summary of the DICTIONARY messages that may accompany Common Data Dictionary messages. Linker messages are described in the *VMS System Messages and Recovery Procedures Reference Volume.*

F.1 Diagnostic Messages from the VAX FORTRAN Compiler

A diagnostic message issued by the compiler describes the detected error and, in some cases, contains an indication of the action taken by the compiler in response to the error.

Besides reporting errors detected in source program syntax, the compiler issues messages indicating errors that involve the compiler itself, such as I/O errors.

F.1.1 Source Program Diagnostic Messages

The severity-level classes of source program diagnostic messages, in order of greatest to least severity, are as follows:

Code	Description
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; should be corrected. An object file is produced despite the E-class error, but the output or program result may be incorrect.
W	Warning; should be investigated by checking the statements to which W-class diagnostic messages apply. Warnings are issued for statements that use acceptable, but nonstandard, syntax and for statements corrected by the compiler. An object file is produced, but the program results may be incorrect. Note that W-class messages are produced unless the /NOWARNINGS qualifier is specified in the FORTRAN command.
I	Information; not an error message and does not call for corrective action. However, the I-class message informs you that either a correct VAX FORTRAN statement may have unexpected results or you have used a VAX extension to FORTRAN-77.

Typing mistakes are a likely cause of syntax errors; they can cause the compiler to generate misleading diagnostic messages. Beware especially of the following:

- Missing comma or parenthesis in a complicated expression or FORMAT statement.
- Misspelled variable names. The compiler may not detect this error, so execution can be affected.
- Inadvertent line continuation mark. This can cause a diagnostic message for the preceding line.
- Extension of the statement line past column 72. This can cause diagnostic messages because the statement is terminated early.
- Confusion between the digit 0 and the uppercase letter O. This 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, nonprinting 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 the question mark cannot occur in a FORTRAN statement, a syntax error usually results.

Because a diagnostic message indicates only the immediate cause, you should always check the entire source statement carefully.

The following examples show how source program diagnostic messages are displayed in interactive mode on your screen. Example F-1 shows how these messages appear in listings.

```
%FORT-W-FMTEXTCOM, Extra comma in format list
        [FORMAT (I3,)] in module MORTGAGE at line 13

%FORT-F-UNDSTALAB, Undefined statement label
        [66] in module MORTGAGE at line 19

%FORT-F-ENDNOOBJ, DB1:[SMITH]MOR.FOR;1 completed
with 2 diagnostics - object deleted
```

Table F-1 is an alphabetical list of FORTRAN diagnostic error messages. For each message, the table gives a mnemonic, an error code level, the text of the message, and an explanation of the message.

Example F-1: Sample Diagnostic Messages (Listing Format)

```
0001    C       Program to calculate monthly mortgage payments
0002
0003            PROGRAM MORTGAGE
0004
0005            TYPE 10
0006    10      FORMAT (' ENTER AMOUNT OF MORTGAGE ')
0007            ACCEPT 20, IPV
0008    20      FORMAT (I6)
0009
0010            TYPE 30
0011    30      FORMAT (' ENTER LENGTH OF MORTGAGE IN MONTHS ')
0012            ACCEPT 40, IMON
0013    40      FORMAT (I3,)
0014
%FORT-W-FMTEXTCOM, Extra comma in format list
        [FORMAT (I3,)] in module MORTGAGE at line 13

0015            TYPE 50
0016    50      FORMAT (' ENTER ANNUAL INTEREST RATE ')
0017            ACCEPT 60, YINT
0018    60      FORMAT (F6.4)
0019            GO TO 66
0020    65      YI = YINT/12   !Get monthly rate
0021            IMON = -IMON
0022            FIPV = IPV * YI
0023            YI = YI + 1
0024            FIMON = YI**IMON
0025            FIMON = 1 - FIMON
0026            FMNTHLY = FIPV/FIMON
0027
0028            TYPE 70, FMNTHLY
0029    70      FORMAT (' MONTHLY PAYMENT EQUALS ',F7.3 )
0030            STOP
0031            END
%FORT-F-UNDSTALAB, Undefined statement label
        [66] in module MORTGAGE at line 19
```

Table F–1: Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
ADJARRBOU	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.
ADJARRUSE	F	Adjustable array used in invalid context A reference to an adjustable array was made in a context where such a reference is not allowed.
ADJLENUSE	F	Passed-length character name used in invalid context A reference to a passed-length character array or variable was made in a context where such a reference is not allowed.
AGGREFSIZ	F	Aggregate reference exceeds 65535 bytes Any aggregate reference larger than 65535 bytes cannot be used in an I/O list or as an actual or dummy argument.
ALTRETLAB	F	Alternate return label used in invalid context An alternate return argument cannot be used in a function reference.
ALTRETOMI	E	Alternate return omitted in SUBROUTINE or ENTRY statement An asterisk was missing in the argument list of a subroutine for which an alternate return was specified. Examples:

•

```
SUBROUTINE XYZ(A,B)
    .
    .
    .
RETURN 1
```

•

```
ENTRY ABC(Q,R)
    .
    .
    .
RETURN I+4
```

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation

ALTRETSPE	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:
		• `INTEGER FUNCTION TCB(ARG,*,X)`
		• `FUNCTION IMAX`
		`.`
		`.`
		`.`
		`RETURN I**`
		`END`

ARIVALREQ	F	Character expression where arithmetic value required An expression that must be arithmetic (INTEGER, REAL, LOGICAL, or COMPLEX) was of type CHARACTER.
ASSARRUSE	F	Assumed size array name used in invalid context An assumed-size array name was used in a context in which the size of the array was required, for example, in an I/O list.
ASSDOVAR	W	Assignment to DO variable within loop The control variable of a DO loop was altered within the range of a DO statement.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation

ATTRIERR	I	COMMON attributes conflict, using the default attribute
		This error only occurs in conjunction with the CDEC$ PSECT compiler directive statement and under any of the following circumstances:
		• A common block is declared as both GBL (global) and LCL (local), both WRT (write) and NOWRT (nowrite), or both SHR (shared) and NOSHR (noshared).
		• More than one alignment (ALIGN=) to the COMMON block is specified.
		• The following combination of compiler directive statements occurs:
		`CPAR$ SHARED com_blk`
		and
		`CDEC$ PSECT /com_blk/ ATTRI=something-not-page-alignment`
		• An alignment value exceeding the legal range is specified. The alignment attribute can only take the value of 0 through 9.

BADEND	F	END [STRUCTURE|UNION|MAP] must match top A STRUCTURE, UNION, or MAP statement did not have a corresponding END STRUCTURE, END UNION, or END MAP statement, respectively.
BADFIELD	F	Field name not defined for this structure A field name not defined in a structure was used in a qualified reference.
BADRECREF	F	Aggregate reference where scalar reference required An aggregate reference was used where a scalar reference was required.
CDDALNARY	I	CDD description specifies an aligned array (unsupported) The CDD description contained an array field whose elements have an alignment that VAX FORTRAN cannot accommodate. When this error is encountered, the array is replaced by a structure of the appropriate size.

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
CDDBITSIZ	F	CDD field specifies a bit size or alignment. Size or address rounded up to byte alignment. The CDD's bit datatype and bit alignment are not supported by VAX FORTRAN.
CDDERROR	I	CDD description extraction condition The VAX FORTRAN compiler was in the process of extracting a data definition from the Common Data Dictionary when an error occurred. See the accompanying messages for more information.
CDDINIVAL	I	CDD description contains Initial Value attribute (ignored) A field that specified an initial value was present in the CDD description being expanded. When this error is encountered, the initial value is ignored.
CDDNOTSTR	F	CDD record is not a structure A CDD record description was not structured. VAX FORTRAN requires structure definitions (elementary field descriptions in CDDL).
CDDRECDIM	F	CDD record is dimensioned VAX FORTRAN does not support dimensioned structures, for example, arrays of structures.
CDDSCALED	W	CDD description specifies a scaled data type VAX FORTRAN does not support scaled data types. The data described by the CDD specified a scaled component.
CDDTOOBIG	E	Attributes for some member of CDD record description exceed implementation's limit for member complexity Some member of the CDD record description had too many attributes and created a program that was too large. Change the CDD description to make the field description smaller.
CDDTOODEEP	E	Attributes for CDD record description exceed implementation's limit for record complexity The CDD record description contained structures that were nested too deeply. Modify the CDD description to reduce the level of nesting in the record description.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation

CHANAMINC	E	Character name incorrectly initialized with numeric value
		A character data item with a length greater than one was initialized with a numeric value in a DATA statement. Example:
		`CHARACTER*4 A`
		`DATA A/14/`

CHASBSLIM	F	Character substring limits out of order The first character position of a substring expression was greater than the last character position. Example: `C(5:3)`
CHAVALREQ	F	Arithmetic expression where character value required An expression that must be of type CHARACTER was of another data type.
COLMAJOR	F	CDD description specifies that it is not a column major array FORTRAN supports only column-major arrays. Change the CDD description to specify a column-major array.
COMVARDECL	F	Common variable cannot be declared CONTEXT_SHARED or PRIVATE A variable within a common block cannot be specified in a CONTEXT_SHARED or PRIVATE compiler directive statement. Entire common blocks can be declared shared or private, but individual elements within them cannot be declared context-shared or private.
CONMEMEQV	E	Conflicting memory attributes in an equivalenced group Through the use of an EQUIVALENCE statement, certain memory locations were given conflicting memory attributes (shared or context-shared and private).
CONSIZEXC	E	Constant size exceeds variable size in data initialization A constant used for data initialization was larger than its corresponding variable.
DBGOPT	I	The NOOPTIMIZE qualifier is recommended with the DEBUG qualifier Optimizations performed by the compiler can cause several different kinds of unexpected behavior when using VMS Debugger. See Chapter 11 for more information.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
DEFSTAUNK	I	Default STATUS='UNKNOWN' used in OPEN statement The OPEN statement default STATUS='UNKNOWN' may cause an old file to be inadvertently modified.
DEPENDITEM	I	CDD description contains Depends Item attribute (ignored) VAX FORTRAN does not support CDD's Depends Item attribute. No action is required.
DICTABORT	F	DICTIONARY processing of CDD record description aborted The VAX FORTRAN compiler was unable to process the CDD record description. See the accompanying messages for further information.
DIRSTRREQ	I	Directive requires string constant, directive ignored This error only occurs in conjunction with the use of the CDEC$ compiler directive statements: TITLE, SUBTITLE, and IDENT. String values for the TITLE, SUBTITLE, and IDENT directives cannot be more than 31 characters. Any other values, including PARAMETER statement constants that are defined to be strings, are invalid on these directives.
ENTDUMVAR	F	ENTRY dummy variable previously used in executable statement The dummy arguments of an ENTRY statement were not used in a previous executable statement in the same program unit.
EQVEXPCOM	F	EQUIVALENCE statement incorrectly expands a common block A common block cannot be extended beyond its beginning by an EQUIVALENCE statement.
EXCCHATRU	E	Non-blank characters truncated in string constant A character or Hollerith constant was converted to a data type that was not large enough to contain all of the significant characters.
EXCDIGTRU	E	Non-zero digits truncated in hex or octal constant An octal or hexadecimal constant was converted to a data type that was not large enough to contain all the significant digits.
EXCNAMDAT	E	Number of names exceeds number of values in data initialization The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. When a mismatch occurs, any extra variables or array elements are not initialized.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXCVALDAT	E	Number of values exceeds number of names in data initialization The number of variables or array elements to be initialized must match the number of constants specified in data initialization. When a mismatch occurs, any extra constant values are ignored.
EXPSTAOVE	F	Compiler expression stack overflow An expression was too complex or too many actual arguments were included in a subprogram reference. A maximum of 255 actual arguments can be compiled. You can subdivide a complex expression or reduce the number of arguments.
EXTARYUSE	I	Extension to FORTRAN–77: Nonstandard use of array One of the following extensions was detected: • An array was used as a FILE specification in an OPEN statement. • The file name of an INQUIRE statement was a numeric scalar reference or a numeric array name reference
EXTCATDARG	I	Extension to FORTRAN–77: Concatenation of dummy argument A character dummy argument appeared as an operand in a concatenation operation.
EXTCHAFOL	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.
EXTCHARREQ	I	Extension to FORTRAN–77: Character required A character variable was initialized with a noncharacter value by means of a DATA statement.
EXTDATACOM	I	Extension to FORTRAN–77: Nonstandard DATA initialization One of the following extensions occurred: • An element in a blank common block was data initialized. • An element of a named common block was data initialized outside of the BLOCK DATA program unit.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXTDATORD	I	Extension to FORTRAN–77: DATA statement out of order A DATA statement occurred prior to a declaration statement. All DATA statements must occur after the declaration section of a program.
EXTILBRNCH	I	Extension to FORTRAN–77: Nonstandard branch into block A nonstandard branch into a DO loop or IF block was detected.
EXTILDOCNT	I	Extension to FORTRAN–77: Negative implied-Do iteration count The iteration count of an implied DO was negative.
EXTMISSUB	I	Extension to FORTRAN–77: Missing array subscripts Only one subscript was used to reference a multi-dimensional array in an EQUIVALENCE statement.
EXTMIXCOM	I	Extension to FORTRAN–77: Mixed numeric and character elements in common A common block must not contain both numeric and character data.
EXTMIXEQV	I	Extension to FORTRAN–77: Mixed numeric and character elements in EQUIVALENCE A numeric variable or numeric array element cannot be equivalenced to a character variable or character array element.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXTOPERAT	I	Extension to FORTRAN–77: Nonstandard operation One of the following operations was detected: • A logical operand and a nonlogical operand were used in the same operation. • A real type expression and a complex type expression were used in the same statement. • A character operand and a noncharacter operand were used in the same operation. • A nonlogical expression was assigned to a logical variable. • A noncharacter expression was assigned to a character variable. • A character dummy argument appeared in a concatenation operation and the result of the expression was not assigned to a character variable. • Logical operators were used with nonlogical operands. • Arithmetic operators were used with nonnumeric operands.
EXTRECUSE	I	Extension to FORTRAN–77: Nonstandard use of field reference A record reference (for example, record-name.field-name) was used in a program compiled with the /STANDARD=[SYNTAX|ALL] qualifier on the FORTRAN command line.
EXTUNDFUNC	I	Extension to FORTRAN–77: Function or Entry name undefined A value was not assigned to either the function name or the entry point name within the body of the function.
EXT_COM	I	Extension to FORTRAN–77: nonstandard comment FORTRAN–77 allows only the characters "C" and "*" to begin a comment line; "c", "D", "d", and "!" are extensions to FORTRAN–77.

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXT_CONST	I	Extension to FORTRAN-77: nonstandard constant The following constant forms are extensions to FORTRAN-77:
EXT_DOEXPR	I	Extension to FORTRAN-77: Nonstandard loop expression The upper bound expression, lower bound expression, or increment expression of a DO loop was not of type integer, real, or double precision.
EXT_FMT	I	Extension to FORTRAN-77: nonstandard FORMAT statement item The following format field descriptors are extensions to FORTRAN-77:

Constant forms (EXT_CONST):

Form	Example
Hollerith	nH.....
Typeless	'xxxx'X or 'oooo'O
Octal	"oooo or Ooooo
Hexadecimal	Zxxxx
Radix-50	nR.....
Complex with PARAMETER components	
COMPLEX*16	(www.xxxDn, yyy.zzzDn)
REAL*16	yyy.zzzQn

Format field descriptors (EXT_FMT):

$,O,Z	All forms
A,L,I,F,E,G,D	Default field width forms
P	Without scale factor

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXT_INTREQ	I	Extension to FORTRAN–77: Integer expression required One of the following items was not of type integer: • A logical unit number • The record specifier, REC=recspec • The arithmetic expression of a computed GOTO statement • The RETURN [I] • A subscript expression • Array dimension bounds • Character substring bounds expressions
EXT_KEY	I	Extension to FORTRAN–77: nonstandard keyword A nonstandard keyword was used.
EXT_LEX	I	Extension to FORTRAN–77: nonstandard lexical item One of the following nonstandard lexical items was used: • An alternate return specifier with an ampersand (&) in a CALL statement • The apostrophe (′) form of record specifier in a direct access I/O statement • A variable format expression
EXT_LOGREQ	I	Extension to FORTRAN–77: Logical expression required One of the following syntax extensions was detected: • A numeric expression was used in a logical IF statement. • A numeric expression was used in a block IF statement. • A value other than .TRUE. or .FALSE. was assigned to a logical variable. • A logical variable was initialized with a nonlogical value by means of a DATA statement.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
EXT_NAME	I	Extension to FORTRAN–77: nonstandard name A name longer than six characters or one that contained a dollar sign ($) or an underscore (_) was used.
EXT_OPER	I	Extension to FORTRAN–77: nonstandard operator The operators .XOR., %VAL, %REF, %DESCR, and %LOC are extensions to FORTRAN–77. The standard form of .XOR. is .NEQV. The % operators are extensions provided to allow access to non-FORTRAN parts of the VMS environment.
EXT_RETTYP	I	Extension to FORTRAN–77: Nonstandard function return type One of the following conditions was detected: • The function was not declared with a standard data type. • The entry point was not declared with a standard data type.
EXT_SOURC	I	Extension to FORTRAN–77: tab indentation or lowercase source The use of tab indentation or lowercase letters in source code is an extension to FORTRAN–77.
EXT_STMT	I	Extension to FORTRAN–77: nonstandard statement type A nonstandard statement type was used.
EXT_SYN	I	Extension to FORTRAN–77: nonstandard syntax One of the following syntax extensions was specified: • `PARAMETER name = value`—Error: No parentheses • `type name/value/`—Error: Data initialization in type declaration • `DATA (ch(exp:exp),v=e2)/values/`—Substring initialization with implied-DO in DATA statement • `CALL name(arg2,,arg3)`—Error: Null actual argument • `READ ( . . . ),iolist`—Error: Comma between I/O control and element lists • `PARAMETER (name2=ABS(name1))`—Error: Function use in PARAMETER statement • `e1 ** -e2`—Error: Two consecutive operators

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation	
EXT_TYPE	I	Extension to FORTRAN-77; nonstandard data type specification The following DATA type specifications are extensions to FORTRAN-77. The FORTRAN-77 equivalent is given where available. This message is issued when these types are used in the IMPLICIT statement or in a numeric type statement.	
		Extension	**Standard**
		BYTE	-
		LOGICAL*1	-
		LOGICAL*2	LOGICAL (with /NOI4 specified only)
		LOGICAL*4	LOGICAL
		INTEGER*2	INTEGER (with /NOI4 specified only)
		INTEGER*4	INTEGER
		REAL*4	REAL
		REAL*8	DOUBLE PRECISION
		REAL*16	-
		COMPLEX*8	COMPLEX
		COMPLEX*16	-
		DOUBLE COMPLEX	-
FLDNAME	F	Structure field is missing a field name Unnamed fields are not allowed. The effect of an unnamed field can be achieved by using %FILL in place of a field name in a typed data declaration.	
FMTEXTCOM	W	Extra comma in format list A format list contained an extra comma. Example: `FORMAT (I4,)`	
FMTEXTNUM	E	Extra number in format list A format list contained an extraneous number. Example: `FORMAT (I4,3)`	
FMTINVCHA	E	Format item contains meaningless character An invalid character or a syntax error was detected in a FORMAT statement.	

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
FMTINVCON	E	Constant in format item out of range A numeric value in a FORMAT statement exceeds the allowable range. Refer to the *VAX FORTRAN Language Reference Manual* for information about range limits.
FMTMISNUM	E	Missing number in format list An expected number was missing from a format list. Example: `FORMAT (F6.)`
FMTMISSEP	E	Missing separator between format items A required separator character was omitted between fields in a FORMAT statement.
FMTNEST	E	Format groups nested too deeply Format groups cannot be nested beyond eight levels.
FMTPAREN	E	Unbalanced parentheses in format list The number of right parentheses must match the number of left parentheses.
FMTSIGN	E	Format item cannot be signed A signed constant is valid only with the P format code.
HOLCOURED	E	Count of Hollerith or Radix–50 constant too large, reduced The value specified by the integer preceding the H or R was greater than the number of characters remaining in the source statement.
IDOINVOP	F	Invalid operation in implied-DO list An invalid operation was attempted in an implied-DO list in a DATA statement, for example, a function reference in the subscript or substring expression of an array or character substring reference. Example: `DATA (A(SIN(REAL(I))), I=1,10) /101./`
IDOINVPAR	F	Invalid DO parameters in implied-DO list An invalid control parameter was detected in an implied-DO list in a DATA statement, for example, an increment of zero.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
IDOINVREF	F	Invalid reference to name in implied-DO list A control parameter expression in an implied-DO list in a DATA statement contained a name that was not the name of a control variable within the scope of any implied-DO list. Example: `DATA (A(J), J=1,10),(B(I), I=J,K) /1001./` Both J and K in the second implied-DO list are invalid names.
IDOSYNERR	F	Syntax error in implied-DO list in data initialization Improper syntax was detected in an implied-DO list in data initialization, for instance, improperly nested parentheses.
ILBRANCH	E	Illegal branch into or out of parallel DO-loop A branch into or out of a parallel DO loop is not allowed.
ILDIRSPEC	E	Illegal directive specification A directive (either CPAR$ or CDEC$) was detected in the first 5 columns of a source code statement. The remainder of the directive contains illegal syntax.
ILDOPARCTL	F	Illegal parallel DO-loop, control variable must be declared INTEGER Only integer control variables can be used with parallel DO loops.
ILDOPARDIR	E	DO_PARALLEL directive must be followed by DO statement, directive ignored The first executable statement after a DO_PARALLEL compiler directive statement (CPAR$ DO_PARALLEL) must be a DO statement.
ILPARSTMT	E	Statement not permitted inside parallel DO-loop I/O statements and RETURN, STOP, and PAUSE statements are not permitted inside a parallel DO-loop.
IMPDECLAR	W	Use of implicit with declaration warnings An IMPLICIT statement was used in a program compiled with the /WARNINGS=DECLARATIONS qualifier on the FORTRAN command line.
IMPMULTYP	E	Letter mentioned twice in IMPLICIT statement, last type used A letter was given an implicit data type more than once. When this error is encountered, the last data type given is used.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
IMPNONE	E	Untyped name, must be explicitly typed The displayed name was not defined in any data type declaration statement, and an IMPLICIT NONE statement was specified. Check that the name was not accidentally created by an undetected syntax error. Example: `DO 10 I = 1.10` The apparent DO statement is really an assignment to the accidentally created variable DO10I.
IMPSYNERR	E	Syntax error in IMPLICIT statement Improper syntax was used in an IMPLICIT statement.

INCDONEST	F	DO or IF statement incorrectly nested
		One of the following conditions was encountered:
		• A statement label specified in a DO statement was used previously. Example:
		`10    I = I + 1`
		`      J = J + 1`
		`      DO 10 K=1,10`

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
		• A DO loop contains an incomplete DO loop or IF block. Examples: (see code below) The start of the incomplete IF block can be a block IF, ELSE IF, or ELSE statement.

```
    DO 10 I=1,10
       J = J + 1
       DO 20 K=1,10
          J = J + K
10  CONTINUE
```

```
    DO 10 I=1,10
       J = J + I
       IF (J .GT. 20) THEN
          J = J - 1
       ELSE
          J = J + 1
10  CONTINUE
       END IF
```

Mnemonic	Error Code	Text/Explanation
INCFILNES	F	INCLUDE files and/or DICTIONARY statements nested too deeply Up to 10 levels of nested INCLUDE files and DICTIONARY statements are permitted.
INCFUNTYP	F	Inconsistent function data types The function name and entry points in a function subprogram must be consistent within one of three groups of data types: Group 1: All numeric types except REAL*16, COMPLEX*16 Group 2: REAL*16, COMPLEX*16 Group 3: Character Example:

```
CHARACTER*15 FUNCTION I
REAL*4 G
ENTRY G
```

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation

INCLABUSE	F	Inconsistent usage of statement label
		
		Labels of executable statements were confused with labels of FORMAT statements or with labels of nonexecutable statements. Example:
		
		`    GO TO 10`
		`10  FORMAT (I5)`

INCLENMOD	F	Incorrect length modifier in declaration An unacceptable length was specified in a data type declaration, for example: `INTEGER PIPES*8`
INCMODNAM	F	Module name not found in library When an INCLUDE statement of the form `INCLUDE '(module)'` is used, several text libraries are searched for the specified module name. These are, in order: 1. Libraries specified on the FORTRAN command line with the /LIBRARY qualifier 2. The library specified using the logical name FORT$LIBRARY 3. The VAX FORTRAN system text library, SYS$LIBRARY:FORSYSDEF. The INCMODNAM message is issued when the specified module name cannot be found in any of the libraries. Note that one of the causes of this search failure may be an open failure on one of the libraries. If a "$LIBRARY/LIST" command shows the module to be present in the library, check to ensure that the library itself can be read by the compiler.
INCOMPNSYSL	W	Unable to open system definition text library SYS$LIBRARY:FORSYSDEF.TLB In an attempt to include a text library, the compiler was unable to open the FORTRAN system definition library.
INCOPEFAI	F	Open failure on INCLUDE file The specified file could not be opened, possibly due to an incorrect file specification, nonexistent file, unmounted volume, or a protection violation.

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
INCOPNFORT	W	Unable to open text library defined by FORT$LIBRARY In an attempt to include a text library, the compiler was unable to open the text library defined by the logical name FORT$LIBRARY.
INCSTAFUN	E	Inconsistent statement function reference The actual arguments in a statement function reference did not agree in either order, number, or data type with the formal arguments declared.
INCSYNERR	F	Syntax error in INCLUDE file specification The file-name string was not acceptable (invalid syntax, invalid qualifier, undefined device, and so on).
INQUNIT	F	Missing or invalid use of UNIT or FILE specifier in INQUIRE statement An INQUIRE statement must have a UNIT specifier or a FILE specifier, but not both.
INTFUNARG	E	Arguments incompatible with intrinsic function, assumed EXTERNAL A function reference was made using an intrinsic function name, but the argument list does not agree in order, number, or type with the intrinsic function requirements. When this error is encountered, the function is assumed to be supplied by you as an EXTERNAL function.
INTVALREQ	F	Non-integer expression where integer value required An expression that must be of type integer was of some other data type.
INVACTARG	E	Invalid use of intrinsic function name as actual argument A generic intrinsic function name was used as an actual argument.
INVASSVAR	E	Invalid ASSOCIATEVARIABLE specification An ASSOCIATEVARIABLE specification in an OPEN or DEFINE FILE statement was a dummy argument or an array element.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
INVCHAUSE	E	Invalid character used in constant An invalid character was detected in a constant. The following characters are valid: 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.
INVCONST	E	Arithmetic error while evaluating constant or constant expression The specified value of a constant was too large or too small to be represented.
INVCONSTR	F	Invalid control structure using ELSE IF, ELSE, or END IF The order of ELSE IF, ELSE, or END IF statements was incorrect. ELSE IF, ELSE, and END IF statements cannot stand alone. ELSE IF and ELSE must be preceded by either a block IF statement or an ELSE IF statement. END IF must be preceded by either a block IF, ELSE IF, or ELSE statement. Examples:

```
DO 10 I=1,10
  J = J + I
  ELSE IF (J .LE. K) THEN
```

Error: ELSE IF preceded by a DO statement.

```
IF (J .LT. K) THEN
    J = I + J
ELSE
    J = I - J
ELSE IF (J .EQ. K) THEN
END IF
```

Error: ELSE IF preceded by an ELSE statement.

Mnemonic	Error Code	Text/Explanation
INVDEVSPE	E	Invalid device specified, analysis data file not produced The file specified by the /ANALYSIS_DATA qualifier could not be written because it was not a random access file.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
INVDOTERM	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, ELSE IF, END IF, DO, or END statement.
INVDUMARG	E	Dummy argument invalid in parallel memory directive Dummy arguments cannot be specified on a parallel memory directive.
INVENDKEY	W	Invalid END= keyword, ignored The END keyword was used illegally in a WRITE, REWRITE, direct access READ, or keyed access READ statement.
INVENTRY	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.
INVEQVCOM	F	Invalid equivalence of two variables in common Variables in common blocks cannot be equivalenced to each other.

INVFUNUSE	F	Invalid use of function name in CALL statement
		A CALL statement referred to a subprogram name that was used as a CHARACTER, REAL*16, or COMPLEX*16 function. Example:
		`IMPLICIT CHARACTER*10(C)`
		`CSCAL = CFUNC(X)`
		`CALL CFUNC(X)`

INVINIVAR	E	Invalid initialization of variable not in common An attempt was made in a BLOCK DATA subprogram to initialize a variable that was not in a common block.
INVINTFUN	E	Name used in INTRINSIC statement is not an intrinsic function A function name that appeared in the INTRINSIC statement was not an intrinsic function.

INVIOSPEC	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`

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
INVKEYOPE	F	Incorrect keyword in OPEN, CLOSE, or INQUIRE statement An OPEN, CLOSE, or INQUIRE statement contained a keyword that was not valid for that statement.
INVLEFSID	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.
INVLEXEME	F	Variable name, constant, or expression invalid in this context An entity was used incorrectly; for example, the name of a subprogram was used where an arithmetic expression was required.
INVLOGIF	F	Statement cannot appear in logical IF statement A logical IF statement must not contain a DO statement or another logical IF, IF THEN, ELSE IF, ELSE, END IF, or END statement.
INVNMLELE	F	Invalid NAMELIST element A dummy argument or element other than variable or array name appeared in a NAMELIST declaration.
INVNUMSUB	F	Number of subscripts does not match array declaration More or fewer dimensions than were declared for the array were referenced.
INVPERARG	F	Invalid argument to %VAL, %REF, %DESCR, or %LOC The argument specified for one of the built-in functions was not valid. Examples: • `%VAL (3.5D0)`—Error: Argument cannot be REAL*8, REAL*16, character, or complex. • `%LOC (X+Y)`—Error: Argument must not be an expression.
INVPERUSE	E	%VAL, %REF, or %DESCR used in invalid context The argument list built-in functions %VAL, %REF, and %DESCR cannot be used outside an actual argument list. Example: `X = %REF(Y)`
INVQUAL	I	Invalid qualifier or qualifier value in OPTIONS statement An invalid qualifier or qualifier value was specified in the OPTIONS statement. When this error is encountered, the qualifier is ignored.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
INVRECUSE	F	Invalid use of record or array name A statement in the program violated one of the following rules: • An aggregate cannot be assigned to a nonaggregate or to an aggregate with a structure that is not the same. • An array name reference cannot be qualified. • Aggregate references cannot be used in I/O lists of formatted I/O statements.
INVREPCOU	E	Invalid repeat count in data initialization, count ignored The repeat count in a data initialization was not an unsigned, nonzero integer constant. When this error is encountered, the count is ignored.

INVSBSREF	E	Substring reference used in invalid context
		A substring reference to a variable or array that is not of type character was detected. Example:
		`REAL X(10)`
		`Y = X(J:K)`

INVSTALAB	W	Invalid statement label ignored An improperly formed statement label (namely, a label containing letters) was detected in columns 1 to 5 of an initial line. When this error is encountered, the statement label is ignored.
INVSUBREF	F	Subscripted reference to non-array variable A variable that is not defined as an array cannot appear with subscripts.
INVTYPUSE	F	Name previously used with conflicting data type A data type was assigned to a name that had already been used in a context that required a different data type.
IODUPKEY	F	Duplicated keyword in I/O statement Each keyword subparameter in an I/O statement or auxiliary I/O statement can be specified only once.

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
IOINVFMT	F	Format specifier in error The format specifier in an I/O statement was invalid. It must be one of the following: • The label of a FORMAT statement • An asterisk (*) in a list-directed I/O statement • A run-time format specifier: variable, array element, or character substring reference • An integer variable that was assigned a FORMAT label by an ASSIGN statement
IOINVKEY	F	Invalid keyword for this type of I/O statement An I/O statement contained a keyword that cannot be used with that type of I/O statement.
IOINVLIST	F	Invalid I/O list element for input statement An input statement I/O list contained an invalid element, such as an expression or a constant.
IOSYNERR	F	Syntax error in I/O list Improper syntax was detected in an I/O list.
LABASSIGN	F	Label in ASSIGN statement exceeds INTEGER*2 range A label whose value is assigned to an INTEGER*2 variable by an ASSIGN statement must not be separated by more than 32K bytes from the beginning of the code for the program unit.

LENCHAFUN	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`

LOGVALREQ	F	Non-logical expression where logical value required An expression that must be of type LOGICAL was of another data type.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
LOG4LCKREQ	E	Lock variable must be declared LOGICAL*4 The lock entity used in a LOCKON or LOCKOFF directive must be declared to be LOGICAL*4.
LOWBOUGRE	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.
MINDIGITS	W	CDD description specifies precision less than allowed for data type. Minimum precision is supplied. Some Common Data Dictionary data types specified a number of digits that is incompatible with VAX FORTRAN data types. When this error is encountered, the VAX FORTRAN compiler expands the data type to conform to a VAX FORTRAN data type. No action required.
MINOCCURS	I	CDD description contains Minimum Occurs attribute (ignored) VAX FORTRAN does not support the CDD's Minimum Occurs attribute. No action required.
MISSAPOS	E	Missing apostrophe in character constant A character constant must be enclosed in apostrophes.
MISSCOM	F	Missing common block name A common block name was omitted or specified improperly on a SHARED directive.
MISSCONST	F	Missing constant A required constant was not found.
MISSDEL	F	Missing operator or delimiter symbol Two terms of an expression were not separated by an operator, or a punctuation mark (such as a comma) was omitted. Examples: • `CIRCUM = 3.14 DIAM` • `IF (I 10,20,30`
MISSEND	E	Missing END statement, END is assumed An END statement was missing at the end of the last input file. When this error is encountered, an END statement is inserted.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
MISSEXPO	E	Missing exponent after E, D, or Q A floating-point constant was specified in E, D, or Q notation, but the exponent was omitted.
MISSKEY	F	Missing keyword A required keyword, such as TO, was omitted from a statement such as ASSIGN 10 TO I.
MISSLABEL	F	Missing statement label A required statement label reference was omitted.
MISSNAME	F	Missing variable or subprogram name A required variable name or subprogram name was not found.
MISSUNIT	F	Unit specifier keyword missing in I/O statement An I/O statement must include a unit specifier subparameter.
MISSVAR	F	Missing variable or constant An expression or a term of an expression was omitted. Examples: • `WRITE ( )` • `DIST = *TIME`
MISSVARCOM	E	Missing variable or common name A name of a variable or a common block that is required by a compiler directive statement or a VOLATILE statement was omitted.
MULDECNAM	F	Multiple declaration of name A name appeared in two or more inconsistent declaration statements.
MULDECTYP	E	Multiple declaration of data type for variable, first type used A variable appeared in more than one data type declaration statement. When this error is encountered, the first type declaration is used.
MULDEFLAB	E	Multiple definition of statement label, second ignored The same label appeared on more than one statement. When this error is encountered, the first occurrence of the label is used.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
MULFLDNAM	F	Multiply defined field name Each field name within the same level of a given structure declaration must be unique.
MULSPEPAR	E	Multiple specification of parallel memory attributes, first specification used A variable, array, record, or COMMON block was a given memory attributes (shared and private or context-shared and private) in a parallel directive. When this error is encountered, the first attribute specified is the one that is used.
MULSTRNAM	F	Multiply defined STRUCTURE name A STRUCTURE name must be unique among STRUCTURE names.
NAMTOOLON	W	Name longer than 31 characters A symbolic name cannot exceed 31 characters. When this error is encountered, the symbolic name is truncated to 31 characters.
NESTPARDO	E	Nested parallel DO-loops not permitted, directive ignored A parallel DO-loop directive (CPAR$ DO_PARALLEL) was detected within a DO-loop that was already marked as parallel. Nested parallel DO-loop directives are not supported.
NMLIOLIST	E	I/O list not permitted with namelist I/O An I/O statement with a namelist specifier incorrectly contained an I/O list.
NODFLOAT	W	CDD description specifies the D_floating data type. The data cannot be represented when compiling /G_FLOAT. A D_floating data type was specified when compiling with the /G_FLOATING qualifier. Ignore the warning message or recompile the program using the /NOG_FLOATING qualifier.
NOGFLOAT	W	CDD description specifies G_floating data type. The data cannot be represented when compiling /NOG_FLOAT. A G_floating data type was specified when compiling with the /NOG_FLOATING qualifier. Ignore the warning message or recompile the program using the /G_FLOAT qualifier.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
NOHFLOAT	W	CDD description specifies H_floating data type. The data cannot be represented when compiling /NOG_FLOAT. An H_floating data type was specified when compiling with the /NOG_FLOATING qualifier. Ignore the warning message or recompile the program using the /G_FLOATING qualifier.
NONCONSUB	F	Nonconstant subscript where constant required Subscript and substring expressions used in DATA and EQUIVALENCE statements must be constants.

NOPATH	W	No path to this statement
		
		Program control could not reach this statement. When this situation occurs, the statement is deleted. Example:
		
		`10    I = I + 1`
		`      GO TO 10`
		`      STOP`

NOSOUFILE	F	No source file specified A command line was entered that specified only library file names and no source files to compile.

OPEDOLOOP	F	Unclosed DO loop or IF block
		
		The terminal statement of a DO loop or the END IF statement of an IF block was not found. Example:
		
		`DO 20 I=1,10`
		`X = Y`
		`END`

OPENOTPER	F	Operation not permissible on these data types An invalid operation was specified, such as an .AND. of two real variables.
PROSTOREQ	F	Program storage requirements exceed addressable memory The storage space allocated to the variables and arrays of the program unit exceeded the addressing range of the machine.
PRVCTLVAR	I	Control variable for parallel loop defaulting to PRIVATE The control variable for a parallel DO loop was not explicitly declared private.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
PRVSYMIL	E	PRIVATE symbol invalid in routine without parallel DO-loop Symbols declared within a routine that does not contain a parallel DO loop cannot be listed in a PRIVATE directive.
REDCONMAR	W	Redundant continuation mark ignored A continuation mark was detected where an initial line is required. When this error is encountered, the continuation mark is ignored.
REFERENCE	I	CDD description contains Reference attribute (ignored) The CDD's Reference attribute is not supported by VAX FORTRAN. No action required.
SAVPRICON	E	PRIVATE variable must not be declared SAVE Symbols cannot be declared in both a PRIVATE directive and a SAVE statement.
SHRCTLVAR	E	Control variable for parallel DO-loops must be declared PRIVATE The control variable for a parallel DO-loop was explicitly declared SHARED. Control variables for parallel DO-loops must be explicitly declared PRIVATE.
SHRNAMLON	E	Shared COMMON name too long, limited to 26 characters The maximum length of a COMMON block name specified in a SHARED compiler directive statement is 26 characters.
SOURCETYPE	I	CDD description contains Source Type attribute (ignored) VAX FORTRAN does not support the CDD's Source Type attribute. No action required.
STAENDSTR	F	Statement not allowed within structure; structure definition closed A statement not allowed in a structure declaration block was encountered. When this situation occurs, the compiler assumes that you omitted one or more END STRUCTURE statements.
STAINVSTR	E	Statement not allowed within structure definition; statement ignored A statement not allowed in a structure declaration block was encountered. Structure declaration blocks can only include the following statements: typed data declaration statements, RECORD statements, UNION/END UNION statements, MAP/END MAP statements, and STRUCTURE/END STRUCTURE statements.

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
STANOTVAL	E	Statement not valid in this program unit, statement ignored A program unit contained a statement that is not allowed; for example, a BLOCK DATA subprogram containing an executable statement.
STAOUTORD	E	Statement out of order, statement ignored The order of statements was invalid. When this error is encountered, the statement found to be out of order is ignored.
STATOOCOM	F	Statement too complex A statement was too complex to be compiled. It must be subdivided into two or more statements.
STRCONTRU	E	String constant truncated to maximum length A character or Hollerith constant can contain up to 2000 characters. A Radix–50 constant can contain up to 12 characters.
STRDEPTH	F	STRUCTUREs/UNIONs/MAPs nested too deeply The combined nesting level limit for structures, unions, and maps is 20 levels.
STRNAME	E	Outer level structure is missing a structure name An outer level STRUCTURE statement must have a structure name in order for a RECORD statement to be able to reference the structure declaration.
STRNOTDEF	F	Structure name in RECORD statement not defined Either a RECORD statement did not contain a structure name enclosed within slashes or the structure name contained in a RECORD statement was not defined in a structure declaration.
SUBEXPVAL	E	Subscript or substring expression value out of bounds An array element beyond the specified dimensions or a character substring outside the specified bounds was referenced.
SUBNOTALL	F	Subqualifier not allowed with negated qualifier A negated qualifier specified on the command line also specified subqualifier values. For example: /NOCHECK=UNDERFLOW

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
TAGVARIAB	I	CDD description contains Tag Variable attribute (ignored) VAX FORTRAN does not support the CDD's Tag Variable attribute. No action required.
TOOMANCOM	F	Too many named common blocks VAX FORTRAN allows a maximun of 250 named common blocks. You must reduce the number of named common blocks.
TOOMANCON	E	Too many continuation lines, remainder ignored Up to 99 continuation lines are permitted, as determined by the /CONTINUATIONS=n qualifier (the default is 19).
TOOMANDIM	E	More than 7 dimensions specified, remainder ignored An array can be defined as having up to seven dimensions.
TOOMANYDO	F	DO and IF statements nested too deeply DO loops and block IF statements cannot be nested beyond 128 levels.
UNDARR	F	Undimensioned array or statement function definition out of order Either a statement function definition was found among executable statements or an assignment statement involving an undimensioned array was found.
UNDSTALAB	F	Undefined statement label A reference was made to a statement label that was not defined in the program unit.
UNSUPPTYPE	I	CDD description specifies an unsupported data type The Common Data Dictionary description for a structure item attempted to use a data type that is not supported by VAX FORTRAN. The VAX FORTRAN compiler makes the data type accessible by declaring it as an inner structure containing a single CHARACTER %FILL field with an appropriate length. Change the data type to one that is supported by VAX FORTRAN or use the VAX FORTRAN built-in functions to manipulate the contents of the field.
VARINCEQV	F	Variable inconsistently equivalenced to itself EQUIVALENCE statements specified inconsistent relationships between variables or array elements. Example: `EQUIVALENCE (A(1), A(2))`

Table F-1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
VAXELNUNS	W	This feature is unsupported on VAXELN The specified VAX FORTRAN feature is not supported on a VAXELN system.
VFUFEANEX	W	This feature is unsupported and non-executable on ULTRIX The program attempted to use a VAX FORTRAN I/O feature that is not available on an ULTRIX system. If the resulting program is run on an ULTRIX system, a run-time error will be issued if this statement is executed. Major VAX FORTRAN features not available on an ULTRIX system include the following: • OPEN and INQUIRE options: — ORGANIZATION= 'RELATIVE' or 'INDEXED' — ACCESS='KEYED' — RECORDTYPE= 'STREAM' or 'STREAM_CR' — KEY — DEFAULTFILE — USEROPEN • I/O statements DELETE, REWRITE, and UNLOCK • Read statement keyword attributes: KEY, KEYEQ, KEYGE, KEYGT, KEYID

Table F–1 (Cont.): Source Program Diagnostic Messages

Mnemonic	Error Code	Text/Explanation
VFUFEAUNS	W	This feature is unsupported on ULTRIX-32 The program attempted to use a VAX FORTRAN I/O feature that is not available on an ULTRIX system. If the resulting program is run on an ULTRIX system, this construct will be ignored. Major VAX FORTRAN features not available on an ULTRIX system include the following: • OPEN statement keywords (and attributes): — DISPOSE= 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT /DELETE' — BUFFERCOUNT — EXTENDSIZE — INITIALSIZE — NOSPANBLOCKS — SHARED • CLOSE statement keywords (and attributes): — DISPOSE= 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT /DELETE' — STATUS
VFUSRCUNA	W	Requested source is not available on ULTRIX The program attempted to use one of the following VAX FORTRAN I/O features that are not available on an ULTRIX system. • The DICTIONARY statement. • The INCLUDE statement for a text module from a library file.
ZERLENSTR	E	Zero-length string The length specified for a character, Hollerith, hexadecimal, octal, or Radix–50 constant must not be zero.

F.1.2 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 FORTRAN command. Printed messages have the form:

```
FORT-F-MNEMONIC, error_text
```

The first line of the message contains the appropriate file specification or keyword involved in the error. The operating system supplies more specific information about the error whenever possible. For example, a file read error might produce the following error message:

```
%FORT-F-READERR, error reading _DBA0:[SMITH]MAIN.FOR;3
-RMS-W-RTB, 512 byte record too big for user's buffer
-FORT-F-ABORT, abort
```

Table F–2 lists the diagnostic messages that report the occurrence of such compiler-fatal errors. Because the exact content of the message depends upon the individual problem, only the first line of the message is provided here. Also, "file-spec" represents placement of the actual file specification in the message, and "keyword-value" represents the specific keyword value.

Table F–2: Compiler-Fatal Diagnostic Messages

I/O Errors
FORT-F-OPENIN, error opening "file-spec" as input
FORT-F-NOSOUFILE, no source file specified
FORT-F-OPENOUT, error opening "file-spec" as output
FORT-F-READERR, error reading "file-spec"
FORT-F-WRITEERR, error writing "file-spec"
FORT-F-CLOSEIN, error closing "file-spec" as input
FORT-F-CLOSEOUT, error closing "file-spec" as output
Command Qualifier Messages
FORT-F-VALERR, specified value is out of legal range
FORT-F-BADVALUE, "keyword-value" is an invalid keyword value
FORT-F-SUBNOTALL, subqualifier not allowed with negated qualifier
Compiler Internal Logic Error
FORT-F-BUGCHECK, internal consistency failure

If you receive the compiler internal logic error, FORT-F-BUGCHECK, you should report both the error and the circumstance in which it occurred to DIGITAL by means of a Software Performance Report (SPR).

F.1.3 Compiler Limits

There are limits to the size and complexity of a single VAX FORTRAN program unit. There are also limits on the complexity of VAX FORTRAN statements. Table F–3 describes some of these limits.

Table F–3: Compiler Limits

Language Element	Limit
Structure nesting	20
DO and block IF statement nesting (combined)	128
Actual number of arguments per CALL or function reference	255
Named common blocks	250
Format group nesting	8
Labels in computed or assigned GO TO list	500
Parentheses nesting in expressions	40
INCLUDE file nesting	10
Continuation lines	99
FORTRAN source line length	132 characters
Symbolic name length	31 characters
Constants	
Character, Hollerith	2000 characters
Radix–50	12 characters
Array dimensions	7
Number of names in a NAMELIST group	250

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 VMS system generation.

F.2 Diagnostic Messages from the VAX Run-Time System

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, problems in array descriptions, and conditions detected by the operating system. Refer to the *VMS Run-Time Library Routines Volume* for more information.

In order of greatest to least severity, the three classes of run-time diagnostic messages are as follows:

Code	Description
F	Severe error; must be corrected. The program cannot complete execution and is terminated when the error is encountered.
E	Error; should be corrected. The program may continue execution, but the output from this execution may be incorrect.
W	Warning; should be investigated. The program continues executing, but output from this execution may be incorrect.

The following example shows how run-time messages are displayed:

```
%FOR-F-ADJARRDIM, adjustable array dimension error
```

Table F–4 is an alphabetical list of run-time diagnostic messages—without the message prefixes FOR, SS, and MTH. (Refer to Table 5–1 for a presentation of the messages in error-number sequence.) For each message, Table F–4 gives a mnemonic, the message number, the class of the message, the message text, and an explanation of the message.

NOTE

> The letter "C" in the column under the heading "SEV" indicates that program execution can continue immediately after the error if a user-written condition handler specifies that execution continue.

Table F–5 is an alphabetical list of diagnostic messages associated with the run-time support for VAX FORTRAN parallel processing. The messages are presented without their prefixes (FOR). For each message, Table F–5 gives a mnemonic, the class of the message, the message text, and an explanation of the message.

Table F–4: Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
ADJARRDIM	93	F,C	adjustable array dimension error Upon entry to a subprogram, one of the following errors was detected during the evaluation of dimensioning information: • An upper-dimension bound was less than a lower-dimension bound. • The dimensions implied an array that was larger than addressable memory.
ATTACCNON	36	F	attempt to access non-existent record One of the following conditions occurred: • A direct access READ, FIND, or DELETE statement attempted to access a nonexistent record from a relative organization file. • A direct access READ or FIND statement attempted to access beyond the end of a sequential organization file. • A keyed access READ statement attempted to access a nonexistent record from an indexed organization file.
BACERR	23	F	BACKSPACE error One of the following conditions occurred: • The file was not a sequential organization file. • The file was not opened for sequential access. (A unit opened for append access may not be backspaced until a REWIND statement is executed for that unit.) • RMS detected an error condition during execution of a BACKSPACE statement.
CLOERR	28	F	CLOSE error An error condition was detected by RMS during execution of a CLOSE statement.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
DELERR	55	F	DELETE error One of the following conditions occurred: • On a direct access DELETE, the file that did not have relative organization. • On a current record DELETE, the file did not have relative or indexed organization, or the file was opened for direct access. • RMS detected an error condition during execution of a DELETE statement.
DUPFILSPE	21	F	duplicate file specifications Multiple attempts were made to specify file attributes without an intervening close operation. One of the following conditions occurred: • A DEFINE FILE statement was followed by another DEFINE FILE statement. • A DEFINE FILE statement was followed by an OPEN statement. • A CALL ASSIGN statement or CALL FDBSET statement was followed by an OPEN statement.

Table F-4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
ENDDURREA	24	F	end-of-file during read One of the following conditions occurred: • An RMS end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification.
ENDFILERR	33	F	ENDFILE error One of the following conditions occurred: • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential or append access. • An unformatted file did not contain segmented records. • RMS detected an error during execution of an ENDFILE statement.
ERRDURREA	39	F	error during read RMS detected an error condition during execution of a READ statement.
ERRDURWRI	38	F	error during write RMS detected an error condition during execution of a WRITE statement.
FILNAMSPE	43	F	file name specification error A file-name specification given to an OPEN, INQUIRE, or CALL ASSIGN statement was not acceptable to RMS.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
FILNOTFOU	29	F	file not found A file with the specified name could not be found during an open operation.
FINERR	57	F	FIND error RMS detected an error condition during execution of a FIND statement.
FLOOVEMAT	88	F,C	floating overflow in math library A floating overflow condition was detected during execution of a math library procedure. The result returned was the reserved operand, −0.
FLOUNDMAT	89	F,C	floating underflow in math library A floating underflow condition was detected during execution of a math library procedure. The result returned was zero.
FLTDIV	73	F,C	arithmetic trap, zero divide During a floating-point or decimal arithmetic operation, an attempt was made to divide by 0.0. If floating-point, the result returned is the reserved operand, −0. If decimal, the result of the operation is unpredictable.
FLTDIV_F	73	F,C	arithmetic fault, zero divide During a floating-point arithmetic operation, an attempt was made to divide by zero.
FLTOVF	72	F,C	arithmetic trap, floating overflow During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type. The result returned was the reserved operand, −0.
FLTOVF_F	72	F,C	arithmetic fault, floating overflow During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type.
FLTUND	74	F,C	arithmetic trap, floating underflow During an arithmetic operation, a floating-point value became less than the smallest representable value for that data type and was replaced with a value of zero.

Table F-4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
FLTUND_F	74	F,C	arithmetic fault, floating underflow During an arithmetic operation, a floating-point value became less than the smallest representable value for that data type.
FORVARMIS	61	F,C	format/variable-type mismatch An attempt was made either to read or write a real variable with an integer field descriptor (I or L), or to read or write an integer or logical variable with a real field descriptor (D, E, F, or G). If execution continued, the following actions occurred: • If I or L, conversion as if INTEGER*4. • If D, E, F, or G, conversion as if REAL*4.
INCFILORG	51	F	inconsistent file organization One of the following conditions occurred: • The file organization specified in an OPEN statement did not match the organization of the existing file. • The file organization of the existing file was inconsistent with the specified access mode; that is, direct access was specified with an indexed organization file, or keyed access was specified with a sequential or relative organization file.
INCKEYCHG	50	F	inconsistent key change or duplicate key A WRITE or REWRITE statement accessing an indexed organization file caused a key field to change or be duplicated. This condition was not allowed by the attributes of the file, as established when the file was created.

Table F-4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
INCOPECLO	46	F	inconsistent OPEN/CLOSE parameters Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow: • READONLY with STATUS='NEW' or STATUS='SCRATCH' • ACCESS='APPEND' with READONLY, STATUS='NEW', or STATUS='SCRATCH' • DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH' • DISPOSE='DELETE' with READONLY
INCRECLEN	37	F	inconsistent record length One of the following occurred: • An attempt was made to create a new relative, indexed, or direct access file without specifying a record length. • An existing file was opened in which the record length did not match the record size given in an OPEN or DEFINE FILE statement.
INCRECTYP	44	F	inconsistent record type The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.
INFFORLOO	60	F	infinite format loop The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
INPCONERR	64	F,C	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 was set to zero.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
INPRECTOO	22	F	input record too long A record was read that exceeded the explicit or default record length specified when the file was opened. To read the file, use an OPEN statement with a RECL value of the appropriate size.
INPSTAREQ	67	F	input statement requires too much data An unformatted READ statement attempted to read more data than existed in the record being read.
INSVIRMEM	41	F	insufficient virtual memory The VAX FORTRAN Run-Time Library attempted to exceed its virtual page limit while dynamically allocating space.
INTDIV	71	F,C	arithmetic trap, integer zero divide During an integer arithmetic operation, an attempt was made to divide by zero. The result of the operation was set to the dividend, which is equivalent to division by one.
INTOVF	70	F,C	arithmetic trap, integer overflow During an arithmetic operation, an integer value exceeded byte, word, or longword range. The result of the operation was the correct low-order part.
INVARGFOR	48	F	invalid argument to FORTRAN Run-Time Library One of the following conditions occurred: • An invalid argument was given to a PDP–11 FORTRAN compatibility subroutine, such as ERRSET. • The VAX FORTRAN compiler passed an invalid coded argument to the Run-Time Library. This can occur if the compiler is newer than the Run-Time Library in use.
INVARGMAT	81	F	invalid argument to math library One of the mathematical procedures detected an invalid argument value.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
INVKEYSPE	49	F	invalid key specification A key specification in an OPEN statement or in a keyed access READ statement was invalid. For example, the key length may have been zero or greater than 255 bytes, or the key length may not conform to the key specification of the existing file.
INVLOGUNI	32	F	invalid logical unit number A logical unit number greater than 99 or less than zero was used in an I/O statement.
INVMATKEY	94	F	invalid key match specifier for key direction A keyed READ used an invalid key match specifier for the direction of that key. Use KEYGE and KEYGT only on ascending keys. Use KEYLE and KEYLT only on descending keys. Use KEYNXT and KEYNXTNE to avoid enforcement of key direction and match specifier.
INVREFVAR	19	F	invalid reference to variable in NAMELIST input The variable in error is shown as "varname" in the message text. One of the following conditions occurred: • The variable was not a member of the namelist group. • An attempt was made to subscript the scalar variable. • A subscript of the array variable was out-of-bounds. • An array variable was specified with too many or too few subscripts for the variable. • An attempt was made to specify a substring of a noncharacter variable or array name. • A substring specifier of the character variable was out-of-bounds. • A subscript or substring specifier of the variable was not an integer constant. • An attempt was made to specify a substring using an unsubscripted array variable.
KEYVALERR	45	F	keyword value error in OPEN statement An improper value was specified for an OPEN or CLOSE statement keyword requiring a value.

Table F-4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
LISIO_SYN	59	F,C	list-directed I/O syntax error The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged.
LOGZERNEG	83	F,C	logarithm of zero or negative value An attempt was made to take the logarithm of zero or a negative number. The result returned was the reserved operand, −0.
MIXFILACC	31	F	mixed file access modes One of the following conditions occurred: • An attempt was made to use both formatted and unformatted operations on the same unit. • An attempt was made to use an invalid combination of access modes on a unit, such as direct and sequential. The only valid combination is sequential and keyed access on a unit opened with ACCESS='KEYED'. • An attempt was made to execute a FORTRAN I/O statement on a logical unit that was opened by a program coded in a language other than FORTRAN.
NO_ CURREC	53	F	no current record A REWRITE or current record DELETE operation was attempted when no current record was defined.
NO_ SUCDEV	42	F	no such device A file specification included an invalid or unknown device name when an OPEN operation was attempted.
NOTFORSPE	1	F	not a FORTRAN-specific error An error occurred in the user program or in the Run-Time Library that was not a FORTRAN-specific error.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
OPEDEFREQ	26	F	OPEN or DEFINE FILE required for keyed or direct access One of the following conditions occurred: • A direct access READ, WRITE, FIND, or DELETE statement specified a file that was not opened with a DEFINE FILE statement or with an OPEN statement specifying ACCESS='DIRECT'. • A keyed access READ statement specified a file that was not opened with an OPEN statement specifying ACCESS='KEYED'.
OPEFAI	30	F	open failure An error was detected by RMS while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is issued when the error condition is not one of the more common conditions for which specific error messages are provided.
OUTCONERR	63	E,C	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. When this situation is encountered, the field is filled with asterisks.
OUTSTAOVE	66	F	output statement overflows record An output statement attempted to transfer more data than would fit in the maximum record size.

Table F-4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
RECIO_OPE	40	F	recursive I/O operation While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted. One of the following conditions may have occurred: • A function subprogram that performs I/O to the same logical unit was referenced in an expression in an I/O list or variable format expression. • An I/O statement was executed at AST level for the same logical unit. • An exception handler (or a procedure it called) executed an I/O statement in response to a signal from an I/O statement for the same logical unit.
RECNUMOUT	25	F	record number outside range A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was created.
REWERR	20	F	REWIND error One of the following conditions occurred: • The file was not a sequential organization file. • The file was not opened for sequential or append access. • RMS detected an error condition during execution of a REWIND statement.
REWRITERR	54	F	REWRITE error RMS detected an error condition during execution of a REWRITE statement.
SEGRECFOR	35	F	segmented record format error An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was created by a program written in a language other than FORTRAN.

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
SIGLOSMAT	87	F,C	significance lost in math library The magnitude of an argument or the magnitude of the ratio of the arguments to a math library function was so large that all significance in the result was lost. The result returned was the reserved operand, −0.
SPERECLOC	52	F	specified record locked A read operation or direct access write, find, or delete operation was attempted on a record that was locked by another user.
SQUROONEG	84	F C	square root of negative value An argument required the evaluation of the square root of a negative value. The result returned was the reserved operand, −0.
SUBRNG	77	F,C	trap, subscript out of range An array reference was detected outside the declared array bounds.
SYNERRFOR	62	F	syntax error in format A syntax error was encountered while the Run-Time Library was processing a format stored in an array or character variable.
SYNERRNAM	17	F	syntax error in NAMELIST input "text" The syntax of input to a namelist-directed READ statement was incorrect. (The part of the record in which the error was detected is shown as "text" in the message text.)
TOOMANREC	27	F	too many records in I/O statement One of the following conditions occurred: • An attempt was made to read or write more than one record with an ENCODE or DECODE statement. • An attempt was made to write more records than existed.
TOOMANVAL	18	F	too many values for NAMELIST variable "varname" An attempt was made to assign too many values to a variable during a namelist-directed READ statement. (The name of the variable is shown as "varname" in the message text.)

Table F–4 (Cont.): Run-Time Diagnostic Messages

Mnemonic	Error Number	Error Code	Text/Explanation
UNDEXP	82	F,C	undefined exponentiation An exponentiation that is mathematically undefined was attempted, for example, 0.**0. The result returned for floating-point operations was the reserved operand, −0, and for integer operations, zero.
UNIALROPE	34	F	unit already open A DEFINE FILE statement specified a logical unit that was already opened.
UNLERR	56	F	UNLOCK error RMS detected an error condition during execution of an UNLOCK statement.
VFEVALERR	68	F,C	variable format expression value error The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of one was assumed, except for a P edit descriptor, for which a value of zero was assumed.
WRONUMARG	80	F	wrong number of arguments An improper number of arguments was used to call a math library procedure.
WRIREAFIL	47	F	write to READONLY file A write operation was attempted to a file that was declared READONLY in the OPEN statement that is currently in effect.

Table F–5: Run-Time Diagnostic Messages for Parallel Processing

Mnemonic	Error Code	Text/Explanation
COMSHRERR	F	Unable to share memory region from x to y The FORTRAN Run-Time Library could not make the specified memory region shared among the processes participating in the parallel processing environment.
DEFVALUSED	I	Default value of xx used for logical name A default value was used for the specified logical name.

Table F–5 (Cont.): Run-Time Diagnostic Messages for Parallel Processing

Mnemonic	Error Code	Text/Explanation
FAIACTCPU	F	Failed to obtain active CPU count The FORTRAN Run-Time Library could not obtain the active CPU count. Thus, it was unable to set up the parallel processing environment.
FAIDCLEXIT	F	Failed to declare an exit handler The FORTRAN Run-Time Library could not declare an exit handler. Thus, it was unable to set up the parallel processing environment.
FAIIDPRC	F	Failed to identify the process The FORTRAN Run-Time Library could not identify the process. Submit a Software Performance Report that describes the conditions leading to the error.
FAIIMAGNAME	F	Failed to obtain image name The FORTRAN Run-Time Library could not obtain the image name. Thus, it was unable to set up the parallel processing environment.
FAIOWNERID	F	Failed to obtain owner process ID The FORTRAN Run-Time Library could not obtain the owner process identification. Thus, it was unable to set up the parallel processing environment.
FAIPRCID	F	Failed to obtain process ID The FORTRAN Run-Time Library could not obtain the process identification. Thus, it was unable to set up the parallel processing environment.
FAIPRCNAME	F	Failed to obtain process name The FORTRAN Run-Time Library could not obtain the process name. Thus, it was unable to set up the parallel processing environment.
FAISHRSTACK	F	Unable to share the stack region from x to y The FORTRAN Run-Time Library could not make the specified stack region shared among processes participating in the parallel processing environment.
FAISUBPRC	F	Failed to create subprocess The FORTRAN Run-Time Library could not get the process identification. Thus, it was unable to set up the parallel processing environment.

Table F–5 (Cont.): Run-Time Diagnostic Messages for Parallel Processing

Mnemonic	Error Code	Text/Explanation
FATINTERR	F	Fatal internal error in the FORTRAN Parallel Processing Run-Time Library The FORTRAN Run-Time Library detected an unrecoverable, inconsistent condition. Submit a Software Performance Report that describes the conditions leading to the error.
INVCOMADR	F	Invalid memory region addresses The FORTRAN Run-Time Library detected invalid starting and ending addresses for a shared memory region. Submit a Software Performance Report that describes the conditions leading to the error.
INVLCLADR	F	Invalid $LOCAL PSECT addresses The FORTRAN Run-Time Library detected invalid starting and ending addresses for a $LOCAL PSECT. Submit a Software Performance Report that describes the conditions leading to the error.
INVLOGNAM	E	Invalid logical name definition A logical name was defined incorrectly.
INVNUMPRC	F	Invalid number of processes The FORTRAN Run-Time Library detected an invalid number of processes. Submit a Software Performance Report that describes the conditions leading to the error.
INVUNWIND	F	Invalid stack unwinding encountered The FORTRAN Run-Time Library detected an invalid attempt to unwind the stack.
LOCALACCESS	F	Subprocess unable to access the shared $LOCAL PSECT A subprocess could not access the shared $LOCAL PSECT.
LOCALSHRERR	F	Unable to share the $LOCAL PSECT The FORTRAN Run-Time Library could not make the $LOCAL PSECT shared among the processes participating in the parallel processing environment.
MEMSHRERR	F	Memory sharing error The FORTRAN Run-Time Library failed to share data among the processes participating in the parallel processing environment.

Table F-5 (Cont.): Run-Time Diagnostic Messages for Parallel Processing

Mnemonic	Error Code	Text/Explanation
NOPARINIT	I	Parallel processing environment was not available The VAX FORTRAN main program was not compiled with /PARALLEL. As a result, the parallel processing environment was not available.
NOTIMPRET	F	Routine not implemented in this version of FORRTL2 An attempt was made to use a routine that is not implemented in this version of the FORTRAN Run-Time Library.
NOTRUNINPP	I	Unable to run the DO-Loop PC: xx in parallel A parallel DO loop with the specified PC address cannot run in parallel.
STACKSHRERR	F	Stack sharing error The FORTRAN Run-Time Library could not make the stack shared among the processes participating in the parallel processing environment.
STKBUFOVR	F	Stack buffer overflow was detected An internal limit on the number of shared stack regions that your program can have was exceeded. Submit a Software Performance Report that describes the conditions leading to the error.
SUBPRCDIED	F	Subprocess PID: xx terminated A subprocess with the specified process ID was terminated.
TOOMANPRC	E	Too many processes, allowed a maximum of 32 processes A limit on the number of processes participating in the FORTRAN parallel processing environment was exceeded. The limit is currently 32.

F.3 DICTIONARY Error Messages

When an error occurs while using the Common Data Dictionary (CDD) (that is, while compiling a DICTIONARY statement), error messages will be generated from one or more of the following sources:

- The FORTRAN compiler, which generates error messages that begin with %FORT. These messages appear in Table F-3.

- The Common Data Dictionary, which generates error messages that begin with %CDD. These messages appear in Appendix D of the *VAX Common Data Dictionary Utilities Reference Manual.* CDDL error messages appear in Appendix C of the *VAX Common Data Dictionary Data Definition Language Reference Manual.*
- The CRX, which generates error messages that begin with %CRX. These messages are listed in this section.

Most CRX messages are related to errors that cannot be corrected by the user. As indicated, submit an SPR to CDD or to the product that created the record description when you receive one of these messages.

The informational messages are related to problems that do not inhibit the production of an object file. They indicate, however, that your results may not be as you had anticipated.

Table F-6: CRX Error Messages

Mneumonic	Error Code	Message	User Action
BADBASE	E	Field description specifies base other than 2 or 10.	Correct the description to be base 2 or 10.
BADCORLEV	E	Record description specifies unsupported core level.	Submit SPR to CDD or to the product that created the description.
BADDIGITS	E	Field description specifies improper number of digits.	Correct the field description to specify the proper number of digits.
BADFORMAT	E	Record description specifies improper record format.	Submit SPR to CDD or to the product that created the description.
BADLENGTH	E	Field description specifies improper length.	Submit SPR to CDD or to the product that created the description.
BADOCCURS	E	Dimension description improperly specifies Minimum Occurs.	Submit SPR to CDD or to the product that created the description.
BADOFFSET	E	Field description specifies improper offset.	Submit SPR to CDD or to the product that created the description.
BADOVERLAY	E	Field description specifies overlay for nonoverlay field.	Submit SPR to CDD or to the product that created the description.

Table F–6 (Cont.): CRX Error Messages

Mneumonic	Error Code	Message	User Action
BADPRTCL	E	Pathname does not designate a node with record protocol.	Correct the pathname.
BADREFER	E	Field description specifies reference for nonpointer field.	Submit SPR to CDD or to the product that created the description.
BADSCALE	E	Field description specifies scale greater than precision.	Correct the precision or scale specified in the field description.
BADSTRIDE	E	Dimension description specifies improper stride.	Submit SPR to CDD or to the product that created the description.
BADTAGVAR	E	Field description specifies tag for nonoverlay field.	Submit SPR to CDD or to the product that created the description.
INITVAL	I	Initial value in field description being ignored.	No action.
LITERALS	I	Literal definitions in record description being ignored.	No action.
MEMBADTYP	E	Field description specifies data type for field with members.	Submit SPR to CDD or to the product that created the description.
NOCONTIN	I	Improper continuation after a noncontinuable condition.	Submit a FORTRAN SPR.
NOCORATT	E	Record description does not specify core level.	Submit an SPR to CDD or to the product that created the description.
NOFORMAT	E	Record description does not specify record format.	Submit SPR to CDD or to the product that created the description.
NOLENGTH	E	Field description does not specify length.	Submit SPR to CDD or to the product that created the description.
NOLOWER	E	Dimension description does not specify lower bound.	Submit SPR to CDD or to the product that created the description.
NOOFFSET	E	Field description does not specify offset.	Submit SPR to CDD or to the product that created the description.

Table F-6 (Cont.): CRX Error Messages

Mneumonic	Error Code	Message	User Action
NOOVERLAY	E	Field description does not specify overlay for overlay field.	Submit SPR to CDD or to the product that created the description.
NOSTRIDE	E	Dimension description does not specify stride.	Submit SPR to CDD or to the product that created the description.
NOTCOMPUT	E	Field definition specifies numeric attributes for nonnumeric data.	Submit SPR to CDD or to the product that created the description.
NOUPPER	E	Dimension description does not specify upper bound.	Submit SPR to CDD or to the product that created the description.
REFERENCE	I	Reference in overlay description being ignored.	No action.
TAGVALUES	I	Tag values in overlay description being ignored.	No action.
UNALIGNED	E	Field description specifies improper field alignment.	Correct the field description to specify the proper alignment.
UNKFACIL	I	Unknown facility specified for record description extraction.	Submit a FORTRAN SPR.

Index

Reader's Comments

VAX FORTRAN
User Manual
AA-D035E-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page　　Description

______　______________________________

______　______________________________

______　______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________ Dept. ____________

Company ______________________ Date ____________

Mailing Address ______________________________

______________________ Phone ____________

-- Do Not Tear - Fold Here and Tape ------------------------------------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ------------------------------------

Cut Along Dotted Line

Reader's Comments

VAX FORTRAN
User Manual
AA–D035E–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:
Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________ Dept. ____________

Company ______________________ Date ____________

Mailing Address ______________________________

______________________ Phone ____________

— Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line