PDP-11 FORTRAN-77 User's Guide

Order Number: AA-V194B-TK

August 1988

This document contains the information necessary to create, link, and execute PDP-11 FORTRAN-77 *programs on a* PDP-11 *processor*. *Programming* information is provided for the RSX-11M/M-PLUS, RSTS/E, and VMS operating systems.

Revision/Update Information:	This revised document supersedes PDP-11 FORTRAN-77 User's Guide, AA-V194A-TK.
Operating System and Version:	RSX-11M Version 4.4 RSX-11M/M-PLUS Version 4.1 RSTS/E Version 9.6 VAX/VMS Version 4.7
Software Version:	FORTRAN-77 Version 5.3

digital equipment corporation
maynard, massachusetts

First Printing, August 1983
Second Printing, August 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The postpaid Reader's Comments forms at the end of this document request the user's critical evaluation to assist in preparing future documentation.

ZK4338

Contents

EXAMPLES

FIGURES

TABLES

Preface

This manual will help programmers create, link, and execute PDP–11 FORTRAN–77 programs under the RSX–11M, RSX–11M/M–PLUS, RSTS/E, and VMS (under AME) operating systems. These operating systems must run on a machine with a Floating Point Processor or a floating-point microcode option.

The PDP–11 FORTRAN–77 language elements are described in the *PDP–11 FORTRAN–77 Language Reference Manual.*

Intended Audience

This manual is intended for programmers who know the fundamental elements and interrelationships of the FORTRAN programming language; a detailed knowledge of the PDP–11 FORTRAN–77 version of FORTRAN is not essential. A detailed knowledge of the host operating system is not essential, but some familiarity is recommended. Whenever a thorough understanding of a specific aspect of an operating system is necessary, you are directed to the appropriate manual for the required additional information.

Structure of This Document

This manual is organized as follows:

- Chapter 1 contains the necessary information to compile, link, and execute a PDP–11 FORTRAN–77 program on RSX–11M/M–PLUS, RSTS/E, and VMS operating systems.
- Chapter 2 provides information about PDP–11 FORTRAN–77 input/output, including details on file characteristics, record structure, and the use of certain OPEN statement keywords.
- Chapter 3 describes the PDP–11 FORTRAN–77 run-time environment, including the calling conventions, error processing, and program section usage.
- Chapter 4 describes PDP–11 FORTRAN–77 implementation concepts, with particular emphasis on data types, generic functions, DO loops, and floating-point data representation.
- Chapter 5 covers programming considerations relevant to typical PDP–11 FORTRAN–77 applications.
- Chapter 6 discusses the use of character data, including character I/O and the character library functions.
- Chapter 7 discusses the use of indexed files and ISAM; an extended example is included.
- Appendixes A through G summarize internal data representation, diagnostic messages, system-supplied functions, compatibility between PDP–11 FORTRAN–77 and other DIGITAL FORTRAN implementations, and language extensions incorporated in PDP–11 FORTRAN–77. Appendix H covers the procedures for reporting software problems.

Associated Documents

The following documents are relevant to FORTRAN–77 programming:

- *PDP–11 FORTRAN–77 Language Reference Manual*
- *PDP–11 FORTRAN–77 Object Time System Reference Manual*
- *PDP–11 FORTRAN–77 Installation Guide/Release Notes*
- *RMS–11 User's Guide*
- *RMS–11 MACRO Reference Manual*
- *RSX–11M/M–PLUS Guide to Program Development*

- *RSX–11M/M–PLUS Task Builder Manual*
- *RSX–11M/M–PLUS Executive Reference Manual*
- *RSTS/E System Manager's Guide*
- *RSTS/E System User's Guide*
- *RSTS/E Task Builder Reference Manual*
- *RSTS/E Programmer's Utilities Manual*
- *VAX–11/RSX–11M User's Guide*
- *VAX–11/RSX–11 Programmer's Reference Manual*

For a complete list of software documents, see the host operating system documentation directory.

Conventions Used in this Document

The following syntactic conventions are used in this manual:

- All references to FORTRAN–77 denote PDP–11 FORTRAN–77, unless otherwise specified.
- Uppercase type is used in text to indicate system commands and command options.
- Lowercase letters are used in syntax specifications and examples to indicate variables; anything that is not a variable (for example, statement names and keywords) appears in uppercase.
- Brackets ([ ]) indicate optional elements within statements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Horizontal ellipses (. . .) indicate that the preceding item(s) can be repeated one or more times.
- "Real" (lowercase) is used to refer to the REAL*4 (REAL), REAL*8 data types as a group; likewise, "complex" (lowercase) is used to refer to COMPLEX*8; "logical" (lowercase) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group; and "integer" (lowercase) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.
- RSX–11 is used as a generic term for the RSX–11M and RSX–11M /M–PLUS operating systems.

- The term FORTRAN-77 in this manual denotes PDP-11 FORTRAN-77. RSX-11 in this manual refers to the RSX-11M and RSX-11M /M-PLUS operating systems.

NOTE

In addition, the following notations denote special nonprinting characters:

Tab character	<TAB>
Space character	#

Chapter 1

Using PDP-11 FORTRAN-77

DIGITAL's PDP-11 FORTRAN-77 consists of two main parts:

- A FORTRAN-77 compiler, that translates a source program into object code.
- A collection of routines (facilities and services) that a program may need while it is executing. This collection of routines is called the Object Time System (OTS).

PDP-11 FORTRAN-77 operates on the RSX-11M, RSX-11M/M-PLUS, RSTS/E, and VMS operating systems.

1.1 Overview

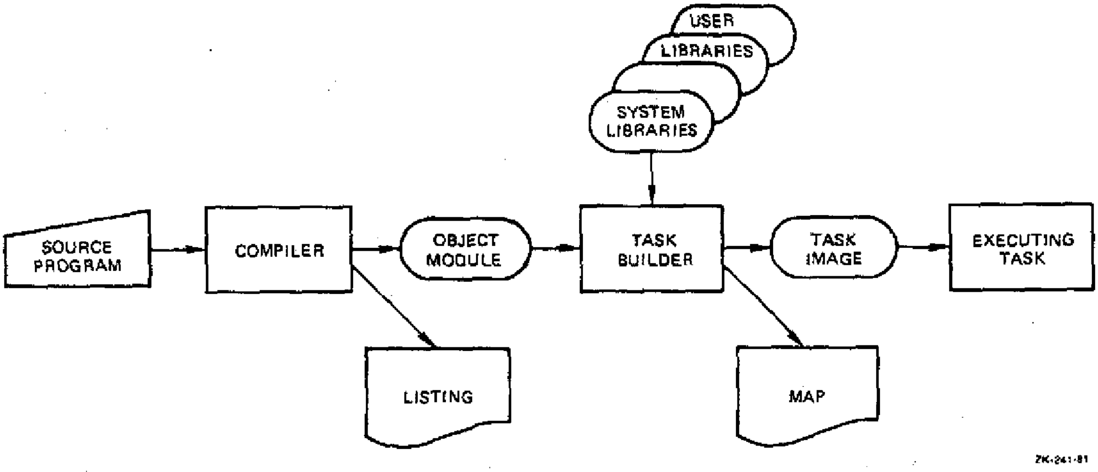
To transform a FORTRAN-77 source program into an executing task, perform these three steps:

1. Compile the program to create a relocatable object module.
2. Task-build the program to link the object module with necessary external routines.
3. Execute the program (and debug it if necessary).

To compile a program, invoke the FORTRAN-77 compiler and specify the source files to be processed; then, task-build it into an executable form called a task image by invoking your system's Task Builder and specifying the object module to be processed. Finally, execute the task image by using the appropriate program execution command for your system.

Figure 1–1 illustrates the process of transforming a FORTRAN–77 source program into an executing task.

Figure 1–1: Preparing a FORTRAN–77 Program for Execution



You invoke the compiler or the Task Builder by entering a command line that specifies the desired function, the input files, the output files, and any desired command options. Command lines are written in one of these command languages: MCR, DCL, or CCL.

You specify input files and output files in command lines using file specifications. File specifications for RSX–11 and VMS system programs differ from those for RSTS/E system programs.

You specify optional command inputs with special command mnemonics called switches. Switches are appended to command words and file specifications.

To efficiently enter a sequence of commands, especially a sequence that is used often, place the sequence in an indirect command file and then type the file name of the indirect command file, preceded by an AT sign (@).

1.2 Using FORTRAN-77 on RSX-11 Systems

The following sections contain information to compile, task-build, and execute a PDP-11 FORTRAN-77 program on an RSX-11M or RSX-11M /M-PLUS system. See Section 1.3 for information on using FORTRAN-77 on RSTS/E systems.

Specifically, the following sections describe how to:

- Write RSX-11 file specifications
- Use command switches
- Use the FORTRAN-77 compiler to create an object module
- Use your system's Task Builder to create a task image
- Execute a task image

1.2.1 RSX-11 File Specifications

For each RSX-11 system program, you must specify the input files to be processed and (optionally for the FORTRAN-77 compiler and your system's Task Builder) the output files to be produced.

The format of a file specification for an RSX-11 system program is as follows:

```
device:[g,m]filename.filetype;version
```

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

[g,m]
The user identification code (UIC) associated with the user file directory containing the desired file. This code consists of a group number (g) and a member number (m). Both g and m are octal numbers. The default value for the UIC is the identification code under which you logged in or where you set your default directory.

[named]
Named directories are supported on RSX-11M and RSX-11M/M-PLUS systems.

filename
The file by its name. A filename value can be up to nine characters long.

filetype
The kind of data in the file. A filetype value can be up to three characters long.

version
The version of the file that is desired. Versions are identified by an octal or decimal number, which is incremented by 1 each time a new version of a file is created.

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

If you request compilation of a source program specified only by a file name, the compiler searches for a file with the specified file name that:

- Is stored on the default device
- Is cataloged under the current default UIC
- Has a file type of .FTN

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

For example, assume that your default device is DK0, that your default UIC is [200,200], and that you supply the following input or output file specification to the compiler:

```
CIRCLE
```

For input, the compiler searches device DK0 in directory [200,200] for the highest version of CIRCLE.FTN. For output, the compiler generates the file CIRCLE.OBJ, stores it on device DK0 in directory [200,200], and assigns it a version that is higher by 1 than any other version of ACIRCLE.OBJ currently cataloged in directory [200,200] on DK0.

Table 1–1: RSX–11 File Specification Defaults

Optional Element	Default Value	
device	User's current default device	
[g,m]	User's current default UIC	
filetype	Depends on usage:	
	Command file	CMD
	Input to compiler	FTN
	Output from compiler	OBJ
	Input to Task Builder	OBJ
	Output from Task Builder	TSK
	Input to RUN command	TSK
	Compiler source listing	LST
	Task Builder map listing	MAP
	Task Builder library input	OLB
	Task Builder overlay description	ODL
	Input to executing program	DAT
	Output from executing program	DAT
version	Input: highest existing version Output: highest existing version plus 1	

1.2.2 Command Switches

Command switches are devices you can use in command lines to specify optional command instructions or inputs—for example, to specify that the compiler compile all lines with a D in column 1.

Command switches are appended to other entities in a command line and have the form:

```
/switch[:val]
```

switch
A mnemonic that specifies a certain instruction to the compiler or Task Builder.

val
A parameter consisting of an octal or decimal number, or a string of characters.

Many switches have a negative form that negates the action specified by the positive form. You can obtain the negative form by following the required slash with a minus sign or the characters NO. For example, /-SP or /NOSP prevents automatic spooling of a program listing.

1.2.3 Compiling a FORTRAN-77 Program with MCR

The PDP-11 FORTRAN-77 compiler is a system program that produces relocatable object modules from FORTRAN-77 source code.

You invoke the FORTRAN-77 compiler with the MCR command F77 as follows:

```
F77 [obj-file] [,list-file] = infiles-list
```

obj-file
The file specification of the object-code output file. You can omit this file specification, if no object file is desired. If it is entered, only a file name value is required. A file type value of OBJ is assumed by default if no file type is specified. Therefore, the following commands are equivalent:

```
F77 FILE1=FILE1

F77 FILE1.OBJ=FILE1
```

Note, that no listing file is created in either case.

list-file
The file specification of the listing output file. You may omit this file specification, if no listing file is wanted. If it is entered, only a file name value is required; a file type value of LST is assumed by default if no file type is specified. Under RSX-11M, the listing file is saved on disk and automatically spooled to the line printer.

infiles-list
The list of input files that contain the source programs. In many cases, this list contains only one file specification; however, when there is more than one, you must separate the individual specifications with commas. Only a file name is normally required; a file type value of FTN is assumed if no file type is specified.

For example, to compile three source programs called WINKN, BLINKN, and NOD into an object module called SINGLE, you would enter:

```
F77  SINGLE, SINGLE = WINKN, BLINKN, NOD
```

or, if you wish:

```
F77  SINGLE.OBJ,SINGLE.LST=WINKN.FTN,BLINKN.FTN,NOD.FTN
```

A F77 command line can also contain one or more of the compiler switches listed and described in Section 1.2.4.

You can also use the F77 command in interactive mode, which permits you to enter multiple compilation commands (lines). To invoke the interactive mode (if you have installed the image of the FORTRAN-77 compiler as F77), you type:

```
F77 RET
```

Regardless of the name under which the PDP-11 FORTRAN-77 compiler is installed, the compiler displays the following prompt:

```
F77>
```

To enter a succession of compilation commands under interactive mode, type one command line after each prompt, followed by a carriage return, until all the commands are entered. Each command line must specify the appropriate input and output files for the program module to be compiled, and any optional switches desired. You then type CTRL/Z. For example, if you want the FORTRAN programs WINKN, BLINKN, and NOD compiled into separate object modules, enter a succession of commands as follows:

```
F77 RET

F77>WINKN,WINKN/SP=WINKN  RET
F77>BLINKN,BLINKN/SP=BLINKN  RET
F77>NOD,NOD/SP=NOD  RET
F77>^Z
```

Note that the compiler displays the F77> prompt each time you enter a command until you type CTRL/Z (^Z) to return system control to MCR.

You can enter the name of an indirect command file in response to the F77> prompt. For example, assume the file COMPILE.CMD contains:

```
WINKN, WINKN/SP=WINKN
BLINKN, BLINKN/SP=BLINKN
NOD, NOD/SP=NOD
```

The following commands are equivalent to the previous example:

```
F77>@COMPILE [RET]
F77>^Z
```

1.2.3.1 Compiling with DCL

You invoke the FORTRAN-77 compiler with the DCL command FORTRAN as follows:

```
FORTRAN [/qualifiers] infiles-list
```

/qualifiers
Optionally included to control the output files and the compiler.

infiles-list
The list of input files containing the source programs to be concatenated and compiled.

/F77
On systems supporting FORTRAN-77 and FORTRAN IV, use the /F77 qualifier to specify FORTRAN-77.

The following DCL qualifiers have no MCR switch equivalents. The remaining DCL qualifiers have effects that are equivalent to the effects of the switches described in Section 1.2.4. Table 1-2 lists the DCL qualifiers and their switch equivalents.

/LIST[:filespec]
Produces a listing file using the file specification provided.

/DS
Generates I- and D-space code. This qualifier does not have an equivalent DCL command.

Table 1-2: DCL Qualifiers and Switch Equivalents

DCL Qualifier	Equivalent Switch
/[NO]CHECK	/[NO]CK
/CONTINUATIONS:n	/CO:n
/[NO]DEBUG	/[NO]DB
/[NO]DLINES	/[NO]DE
none	/DS
/[NO]EXTEND	/[NO]EX
/[NO]F77	/[NO]F77
/IDENTIFICATION	/ID
/[NO]I4	/[NO]I4
/[NO]LIST:filespec	none
/[NO]MACHINE_CODE	/LI:3
/[NO]MAP	/LI:2
/[NO]OBJECT:filespec	none
/[NO]OPTIMIZE	/[NO]OP
/[NO]SHAREABLE	/[NO]RO
/[NO]SOURCE	/LI:2
/[NO]STANDARD[:arg]	/[NO]ST:xxx
ALL	ALL
NONE	NONE
SOURCE	SOURCE
SYNTAX	SYNTAX
/[NO]TRACEBACK:[arg]	/[NO]TR:xxx
ALL	ALL
BLOCKS	BLOCKS
LINES	LINES
NAMES	NAMES
NONE	NONE
/[NO]WARNINGS	/[NO]WR
/WORK_FILES:n	/WF:n

/NOLIST
Does not produce a listing file.

/OBJECT[:filespec]
Produces an object file using the file specification provided.

/NOOBJECT
Does not produce an object file.

1.2.4 Compiler Switches

You use compiler switches to specify optional instructions to the compiler or to specify special attributes for input or output files. A compiler switch consists of a slash followed by a 2-character ASCII name, and has two forms: a positive form and a negative form. For example, if the compiler switch designator is SW, then:

```
/SW sets an action;
/NOSW or /-SW negates that action.
```

Some compiler switches may also be followed by a value. The permitted values are character strings, octal numbers, and decimal numbers. The default radix for a numeric value is decimal. Decimal values may end with a decimal point; octal values always begin with a number sign (#). Some examples of valid compiler switches are:

```
/I4
/TR:NAMES
/CO:25
/CO:#23
```

Append switches that affect listings (for example, /SP, /LI:n) to the specification for the input or output file they will affect. Append all other switches to the FORTRAN-77 command line. Unless the /LA switch is set, all the switches in the following list are initialized to their default values before each compilation.

The compiler switches and their meanings are as follows.

Switch	Description

/CK	Specifies that array references are to be checked to ensure that they are within the array address boundaries specified. However, array upper bounds checking is not performed for arrays that are dummy arguments if the last dimension bound is specified as * or 1. For example:
	`DIMENSION B(0:10,0:*)`
	or
	`DIMENSION A(1)`
	The default setting is /NOCK.

/CO:n	Specifies that the compiler accepts at least n continuation lines. (You may have fewer than n continuation lines.) The value of n may range from 0 to 99; the default value is 19. Each level of nesting of an INCLUDE statement costs two continuation lines.
/DB	Specifies that the compiler is to provide symbol table information for use by the PDP-11 FORTRAN-77 symbolic debugger. When you use the /DB qualifier, you should also use the /NOOP qualifier. Specify the TKB switch /DA when building a program task for debugging. The default setting is /NODB.
/DE	Requests compilation of lines with a D in column one. These lines are treated as comment lines by the default /NODE (see the *PDP-11 FORTRAN-77 Language Reference Manual* for more information).
/DS	Overrides the default compiler's provisions for I- and D-space. These provisions date from installation, when the decision for in-line or I- and D-space code generation was made. (I- and D-space code generation functioned as the default.) When the default compiler generates in-line code, the /DS switch forces I- and D-space code generation. When the default compiler generates I- and D-space code, the /-DS switch forces in-line code generation.
/EX	Specifies that the compiler compile FORTRAN source text that extends up to and includes column 132 of an input record. If /EX is specified, then the ANSI standard extension flagger invoked by the command switch /ST:SOURCE issues an informational diagnostic (one per record) for source lines extending beyond column 72. The default setting is INDEX.

Switch	Description
/F77	Specifies an ANSI X3.9-1978 interpretation at compile time of syntactic and semantic features that have a different interpretation in PDP-11 FORTRAN IV-PLUS Version 3.0. See Appendix E for a detailed discussion of the incompatibilities between PDP-11 FORTRAN-77 and PDP-11 FORTRAN IV-PLUS. The default setting is /F77.
/ID	Types the FORTRAN-77 compiler identification and version number on your terminal. /NOID is the default setting.
/I4	Allocates two words for the default length of integer and logical variables. Normally, single storage words are the default allocation for all integer or logical variables not given an explicit length definition (such as INTEGER*2, LOGICAL*4). /NOI4 is the default setting. See Section 4.2 for more information.
/LA	Causes the current switch settings to be retained (latched) for subsequent compilations in MCR interactive mode. Normally, switch settings are restored to their default values before processing each command line. This switch is convenient for compiling a series of programs in MCR interactive mode with the same switch settings. /NOLA is the default setting.
/LI:n	Specifies listing options. The value of n may range from 0 to 3. The meaning of each value is as follows: n=0 Minimal listing file: diagnostic messages and program section summary only n=1 Source listing and program section summary n=2 Source listing, program section summary, and storage map (default) n=3 Source listing, assembly code, program section summary, and storage map The default setting is /LI:2. See Section 3.6 for a detailed description of the listing format; also refer to the *PDP-11 FORTRAN-77 Object Time System Reference Manual.*
/OP	Directs the compiler to produce optimized code. The negative form, /NOOP, is recommended when /DB is specified. The default setting is /OP.

<table>
<tr><th>Switch</th><th>Description</th></tr>
<tr><td>/RO</td><td>Directs the compiler to specify pure code and pure data sections as read-only in order to take advantage of code sharing in multiuser tasks. See Section 3.3 for a description of program section attributes.
/NORO is the default setting.</td></tr>
<tr><td>/SP</td><td>Requests automatic spooling of the listing file. The default is to spool (/SP).</td></tr>
<tr><td>/ST:xxx</td><td>Directs the compiler to look in your source code for extensions to ANSI standard (X3.9-1978) FORTRAN at the full-language level. If the compiler finds extensions, it flags them and produces informational diagnostics about them. (To receive informational diagnostics, set the warning switch /WR.)
Although PDP–11 FORTRAN–77 conforms to the ANSI FORTRAN standard at the subset level, the compiler flags only those features that are extensions to the full language. See Appendix G for a list of the flagged extensions.
The /ST:xxx switch can take the following forms:
<table>
<tr><td>/ST</td><td>Informational diagnostics for syntax extensions</td></tr>
<tr><td>/ST:ALL</td><td>Informational diagnostics for all detected extensions</td></tr>
<tr><td>/ST:NONE</td><td>No informational diagnostics</td></tr>
<tr><td>/ST:SOURCE</td><td>Informational diagnostics for lowercase letters and tab characters in source code</td></tr>
<tr><td>/ST:SYNTAX</td><td>Same as /ST</td></tr>
<tr><td>/NOST</td><td>Same as /ST:NONE</td></tr>
</table>
The default value is /ST:NONE.
See Section C.2 for a list of compiler diagnostic messages.</td></tr>
</table>

<table>
<tr><th>Switch</th><th>Description</th></tr>
<tr><td>/TR:xxx</td><td>Controls the amount of extra code included in the compiled output for use by the OTS during error traceback. This code is used in producing diagnostic information and in identifying which statement in the source program caused an error during execution. /TR:xxx can have the following forms:
<table>
<tr><td>/TR</td><td>Same as /TR:ALL.</td></tr>
<tr><td>/TR:ALL</td><td>Error traceback information is compiled for all source statements and function and subroutine entries.</td></tr>
<tr><td>/TR:LINES</td><td>Same as /TR:ALL.</td></tr>
<tr><td>/TR:BLOCKS</td><td>Traceback information is compiled for subroutine and function entries and for selected source statements. The source statements selected by the compiler are initial statements in sequences called blocks (see Section 5.2.3 for the definition of a block).</td></tr>
<tr><td>/TR:NAMES</td><td>Traceback information is compiled only for subroutine and function entries.</td></tr>
<tr><td>/TR:NONE</td><td>No traceback information is produced.</td></tr>
<tr><td>/NOTR</td><td>Same as /TR:NONE.</td></tr>
</table>
The default value is /TR:BLOCKS.
Use the setting /TR during program development and testing. Use the default setting /TR:BLOCKS for most programs in regular use. The setting /NOTR may be used for fast execution and minimal code, but it provides no information to the OTS for diagnostic message traceback.</td></tr>
<tr><td>/WF:n</td><td>Determines the number of temporary disk work files to be used during compilation. From one to three files can be used; the default value of n is 2. Increasing the number of files increases the size of the largest program that can be compiled, but may decrease compilation speed.</td></tr>
<tr><td>/WR</td><td>Enables compiler warning diagnostics (W-class messages; see Section C.2.1). If /NOWR is set, no warning messages are issued by the compiler. The default setting is /WR.</td></tr>
</table>

1.2.5 Task-Building a FORTRAN-77 Program

The Task Builder is a system program that links relocatable object modules to form an executable task image. You invoke the Task Builder by entering the MCR command TKB. TKB is described in Section 1.2.5.1.

The object modules to be linked can come from user-specified input files, user libraries, or system libraries. The Task Builder resolves references to symbols defined in one module and referred to in other modules. Should any symbols remain undefined after all user-specified input files are processed, the Task Builder automatically searches the system object library LB:[1,1]SYSLIB.OLB to resolve them.

The default FORTRAN–77 object time system library normally either is part of the system object library or is separate object library LB:[1,1]F77FCS.OLB or LB:[1,1]F77RMS.OLB. Consult your system manager to determine whether the FORTRAN–77 object time system (OTS) is part of SYSLIB.OLB or is a separate library.

Two versions of the OTS I/O support modules for FORTRAN–77 are distributed. One version uses File Control Services (FCS-11), which supports sequential and direct access to sequential files. The other version of the OTS I/O support library uses Record Management Services (RMS–11), which supports sequential, direct, and keyed access to sequential, relative, and indexed files. Consult your system manager to determine which version of the I/O support library is the default on your system and where the other version of the I/O support library is maintained, should you need it.

The FCS-11 file system is always contained in the system object library (that is, in LB:[1,1]SYSLIB.OLB); the RMS–11 file system is always contained in a separate object library (that is, LB:[1,1]RMSLIB.OLB).

The Task Builder also resolves references to resident common blocks and resident libraries; the task image produced, therefore, is ready to be run under the operating system.

You can also use the Task Builder to build tasks with overlay structures. For additional information about the Task Builder and Task Builder options, refer to the Task Builder manual for your operating system.

1.2.5.1 Using the MCR Command TKB

You use the MCR command TKB to invoke the Task Builder.

The TKB command line has the format:

```
TKB [task-file]/FP[,map-file] = infiles-list
```

task-file
The file specification of the task-image output file. This file specification may be omitted if no task-image file is desired. If a specification is entered, only a file name is required; a file type value of TSK is assumed if no file type is specified. Therefore, the commands

```
TKB  FILE1/FP=FILE1
```

and

```
TKB  FILE1.TSK/FP=FILE1
```

are equivalent. Note, however, that no map file is created in either case.

The following switches may be applied to the task-image file:

/FP
Specifies that the task use the Floating Point Processor (FP11) or floating-point microcode option (KEF11A).

NOTE

You *must* include the /FP switch when you build a task; if you do not, the task will exit with the FORTRAN run-time message: "TASK INITIALIZATION FAILURE." (Refer to Section 5.4.1 for the one exception to this rule.)

/DA
Specifies that the system debugging aid ODT is to be included in the task.

/ID
Specifies that the task use I– and D–space. You can build an I– and D–space task on RSX–11M/M–PLUS (Version 2.0 or higher), Micro/RSX (Version 3.0 or higher), RSTS/E (Version 9.0 or higher), and
Micro/RSTS (Version 2.0 or higher).

The default FORTRAN–77 compiler supports I– and D–space. This may cause some tasks to grow slightly.

The RSX-11M/M-PLUS and RSTS/E systems allow you to turn off this support. (The Micro systems do not provide this option.) To turn off I- and D-space support, edit the configuration file during installation. Change the DSPACE parameter in this file to 0.

Then re-task-build the compiler, following the instructions listed in the *PDP-11 FORTRAN-77 Installation Guide*.

/MU

Specifies that multiple versions of the task may be run simultaneously. The read-only portions of the task are shared.

map-file

The file specification of the map output file. This file specification may be omitted if no task-image map file is desired. If a specification is entered, only a file name is required; a file type value of MAP is assumed if no file type is specified. The map file is automatically spooled to the line printer. On some operating systems, the map file is automatically deleted after it is printed.

The following switches may be applied to the map file.

/CR

Specifies that a global cross-reference listing is to be appended to the map file.

/SP

Specifies that the map file is to be spooled to the line printer.

infiles-list

The list of input files that contain compiled FORTRAN-77 object modules. (This list may also contain compiled or assembled libraries and modules that were written in a language other than FORTRAN, such as MACRO.) In many cases, this list contains only one file specification; however, when there is more than one specification, you must separate the individual specifications with commas. Only a file name is normally required; a file type value of OBJ is assumed.

The following switches may be applied to input files.

/LB

Specifies that the input file is to be a library file. See Section 1.2.5.3.

/MP

Specifies that the input file is an overlay description file. See Section 1.5.

For example, to build a task image for the object file SINGLE, created in Section 1.2.3, when the FORTRAN–77 OTS is included in the system object library (SYSLIB.OLB), you enter:

```
TKB SINGLE/FP,SINGLE=SINGLE
```

or, if you wish:

```
TKB  SINGLE.TSK/FP,SINGLE.MAP=SINGLE.OBJ
```

Note that under RSX–11 the map file created by these commands is both saved on disk and spooled to the line printer.

If the FORTRAN–77 OTS routines are contained in a separate library, this library must be explicitly specified in the Task Builder command line. For example:

```
TKB  SINGLE/FP,SINGLE=SINGLE,LB:[1,1]F77FCS/LB
```

NOTE

> When using a separate FORTRAN–77 library, take particular care that object modules from other PDP–11 FORTRAN compilers and OTS routines are not accidentally included in a task being built from FORTRAN–77 object modules. Object modules produced by different PDP–11 FORTRAN compilers must not be combined in a single task.

If the default I/O support library on your system is RMS–11, you must explicitly reference RMSLIB in the task-build command line. The previous example then becomes:

```
TKB SINGLE/FP,SINGLE=SINGLE,LB:[1,1]F77RMS/LB,LB:[1,1]RMSLIB/LB
```

You can also use the TKB command in interactive mode, which permits you to enter multiple-line commands. To enter interactive mode, you simply type:

```
TKB  RET
```

The Task Builder then displays the following prompt:

```
TKB>
```

You may now enter a single command line that indentifies all the input files you want to use to begin the task build, followed by a carriage return. Or you may enter additional input files on as many subsequent lines as you need. When you have entered all your input files, you must type a final line consisting of two slash characters (//), followed by a carriage return (see Section 1.2.5.2 if you are entering any Task Builder options). The double slash signals the Task Builder to begin processing.

1.2.5.2 Task Builder Options

The Task Builder allows numerous options to be specified. Several of these are of particular interest to the FORTRAN–77 user.

To specify options in the MCR command TKB, you must use the Task Builder in interactive mode, and you must terminate command input with a line consisting of a single slash (/) (rather than the double slash described in Section 1.2.5.1). The single slash signals the Task Builder to prompt you, as follows, for option information:

```
ENTER OPTIONS:
TKB>
```

At this point, you can enter as many Task Builder options as you need, one option per line. After you enter each option, the Task Builder automatically prompts you for the next option until you enter a single slash (/) to signal no more options. The Task Builder then proceeds to build the task and to produce any requested output. To exit interactive mode in TKB, enter two slashes (//).

The Task Builder options considered useful to you as a FORTRAN–77 programmer are described below.

ACTFIL—You can declare the total number of input and output files that a task can open simultaneously, and allocate the proper number of buffers, by entering:

```
ACTFIL = n
```

n

The number, in decimal, of files that can be opened simultaneously and the buffers needed to accommodate them. The default value is 4.

Any attempt to open a file or use a logical unit when space is not available for at least one buffer will cause an error at run-time.

The value n includes both explicitly and implicitly opened files.

ASG—You can assign logical unit numbers to physical devices by entering the following:

```
   ASG = dev1:n1:n2:...,dev2:m1:m2:...
```

dev

A physical device name.

n

A valid logical unit number.

m

A valid logical unit number.

The default device assignments are as follows:

```
ASG = SY0:1:2:3:4,TI0:5,CL0:6
```

You can build a cluster library for the FORTRAN-77 OTS on RSX-11M Version 4.1, RSX-11M/M-PLUS Version 2.1, and RSTS/E Version 8.0. Both the FCS and RMS versions of the FORTRAN-77 OTS can be built as a cluster library. See the Task Builder manual for your particular operating system for more information on how to build a cluster library for the FORTRAN-77 OTS.

To use the FORTRAN-77 OTS cluster library, use the TKB option CLSTR as shown in the following example:

```
TKB>PROG/FP=PROG,LB:[1,1]F77FCS/LB:F77VEC,F77FCS/LB
TKB>/
ENTER OPTIONS:
TKB>CLSTR=F7FCLS,FCSRES:RO (for FCS clustered-resident library)
TKB>//
```

RMSRES is the FORTRAN-77 RMS OTS resident library; F7FCLS is the FORTRAN-77 FCS OTS resident library.

To save space, you may link to several shared resident libraries by sharing the same cluster, in the following way:

```
CLSTR=name,name,name:access[:apr]
```

name
The library's symbolic name.

access
Either RO for read-only or RW for read-write.

apr
An integer from 1 through 7 that specifies the first active page register into which the resident library is to be linked.

The F77 resident library can now cluster with either the FCS or RMS resident library, FCSCLS or RMSRES, respectively.

COMMON—If a program is to reference a system global common block, you must declare this intention by specifying:

```
COMMON = name:access[:apr]
```

name
The symbolic name associated with the system global common block.

access
Either RO for read-only or RW for read/write.

apr
An integer from 1 through 7 that specifies the first Active Page Register into which the resident library is to be linked. You can specify apr only when the resident library consists of position-independent code. (FORTRAN–77 resident libraries do not consist of position-independent code.)

The FORTRAN COMMON block with the same name is used to reference the data in the system global common.

EXTTSK—You can allocate additional buffer space for RMS–11 input/output by using the option:

```
EXTTSK = n
```

n
The number, in decimal, of words to allocate. The value assigned by this option may be overridden by the /INC option on the RUN command (see Section 1.2.6).

For information on how to determine the amount of buffer space a program may need, refer to the *RMS-11 MACRO-11 Reference Manual.*

On RSTS/E systems, you can use the EXTTSK option to allocate up to 31K words of memory to a task image (if you have the RSX Emulator in the monitor and your default run-time system is RSX).

The EXTTSK option is more efficient than the ACTFIL option because:

- The amount of space can be more accurately specified.
- The space allocated by EXTTSK does not require disk space in the task-image file.

When you use an operating system that supports the Extend Task system directive, the RMS-11 version of the OTS attempts to extend the buffer space dynamically.

FMTBUF—The default size of the buffer used to contain the internally compiled form of a format specification stored in an array is 64 bytes. You can increase the size of this buffer by entering:

```
FMTBUF = n
```

n
The decimal size, in bytes, of the run-time format compilation buffer.

The total size needed for format compilation is equal to the largest run-time format specification used by the program. For information on how to determine the amount of space needed to store a given format, refer to the *PDP-11 FORTRAN-77 Object Time System Reference Manual.*

GBLPAT—To patch FORTRAN logical unit 0 to a valid system logical unit, use the option:

```
GBLPAT= main-prog:$LUNO:n
```

main-prog
The name of your main program segment.

n
A system logical unit number in the range 1 through 99 (see Section_____ 2.1.3).

LIBR—If a program is to reference a system-shared library, you must specify:

```
LIBR = name:access[:apr]
```

name
The library's symbolic name.

access
Either RO for read-only or RW for read/write.

apr
An integer from 1 through 7 that specifies the first Active Page Register into which the resident library is to be linked. You can specify apr only when the resident library consists of position-independent code (PIC). (FORTRAN-77 resident libraries do not consist of position-independent code.) Libraries are discussed in more detail in Section 1.2.5.3.

MAXBUF—The default maximum record size for input/output is set at 133 (decimal) bytes. You can increase this record size by entering:

```
MAXBUF = n
```

n
The number of bytes (in decimal).

The default generally is adequate for sequential input/output. If sequential, direct, or keyed access operations are performed with records larger than 133 bytes, you must use this option, as follows, to specify the size of the largest record you intend to process.

For formatted records:

```
MAXBUF = RECL
```

For unformatted records:

```
MAXBUF = 4*RECL
```

For segmented records (see Section 2.2.3.3 for a definition of segmented records):

```
MAXBUF = (4*RECL)+2
```

The two extra bytes for segmented records are the segment control bytes (see Section 2.2.3.3).

RESLIB—If a program references a user-shared library, you must specify:

```
  RESLIB= file-spec/access[:apr]
```

file-spec
The file specification of the shared-library task image and symbol-table files.

access
Either RO for read-only or RW for read/write.

apr
An integer from 1 through 7 that specifies the first Active Page Register into which the resident library is to be linked. You can specify apr only when the resident library consists of position-independent code (PIC). (FORTRAN-77 resident libraries do not consist of position-independent code.) Libraries are discussed in more detail in Section 1.2.5.3.

UNITS—The default number of logical units available to a program is 6 (logical units 1 through 6, inclusive). You can set this number smaller or larger at task-build time by entering:

```
UNITS = n
```

n
The number, in decimal, of logical units desired, from 0 to 99.

However, you should be aware that increasing the number of default units available will increase task size. (On RSTS/E systems, you can specify only up to 14 logical units: from 1 through 14.)

The default device and file name associated with a logical-unit number are discussed in Section 2.1.1.

When you need to assign devices to the units you have specified with the UNITS option, use the ASG option discussed earlier in this section. If you need more units than the six provided as the default, you must enter the UNITS option before you make any assignments with ASG.

1.2.5.3 Library Usage on RSX–11 Systems

There are two types of RSX–11 libraries, each of which consists of a collection of object modules: relocatable and resident. A relocatable library is one that the Task Builder can make a physical part of a task image. A resident library is one that the Task Builder can make a logical part of a task image but not a physical part; that is, the Task Builder can link it to a task image but cannot copy it to a task image.

Relocatable Libraries—Relocatable libraries are stored in files on disk. From these libraries, the Task Builder copies object modules into the task image of each task that references those modules. You must tell the Task Builder that an input file is contained in a relocatable library by attaching the switch /LB to the input file specification of the file. If you do not include an extension with the file name of such a specification, the Task Builder assumes .OLB as a default. When the Task Builder encounters a library specification, it includes in the task image being built those modules in the specified library that contain definitions of any currently undefined global symbols.

Resident Libraries—Resident libraries are located in main memory and are shareable: that is, a single copy of each library is used by all tasks that refer to it. You gain access to a resident library by using the LIBR or common option, as described in Section 1.2.5.2.

System Libraries—Each RSX–11 system has a system relocatable library and, in addition, has available to it four system resident libraries.

The system relocatable library is as follows:

```
LB:[1,1]SYSLIB.OLB
```

The Task Builder automatically searches the system relocatable library to see if any undefined global references remain after all the input files have been processed. If the definition of one of these undefined global symbols is found, the appropriate object module is included in the task being built.

Four system resident libraries may be available for use with MCR. Consult your system manager to determine which of the following system resident libraries are available on your system.

- FCSRES—A shared library of commonly used FCS-11 input/output routines.
- RMSRES—A shared library of RMS–11 input/output routines can be built in supervisor mode on RMS–11/M–PLUS systems.
- FCSFSL—A supervisor-mode FCS library.

These system resident libraries are linked to a task by using the Task Builder option, as follows:

```
LIBR = FCSRES:RO
```

or

```
LIBR = RMSRES:RO or RESSUP=RMSRES/SV
```

or

```
RESSUP=FCSFSL/SV
```

User Libraries—Using the Librarian Utility, you can construct your own FORTRAN-77 or assembly language relocatable libraries. You then access these libraries by using the appropriate library switch, as described in preceding sections. Consult the *IAS/RSX-11 Utilities Procedures Manual* for further information on the Librarian Utility.

For example, if MATRIXLIB.OLB is a relocatable library containing matrix manipulation routines and PROG is the object file of a compiled FORTRAN-77 program that calls the matrix routines, you could enter the following command line for the Task Builder:

```
TKB PROG/FP=PROG,MATRIXLIB/LB
```

1.2.6 Executing a FORTRAN-77 Program

To begin task execution once you have built a task image, you enter a RUN command of the form:

```
  RUN filespec[/INC=n]
```

filespec
The file specification of the file containing the task image.

n
The number, in decimal, of words of additional buffer space to allocate for the OTS and file-system buffers. (For information on how to determine the proper size of n, refer to the *RMS-11 Macro Reference Manual.*)

You can end a task before its normal completion by typing CTRL/C (^C), followed by the ABORT command, or you can end execution with a STOP statement. When the STOP statement is executed, the OTS will type a line with the task name and the contents of the display text following STOP.

A task that terminates as a result of a CALL EXIT statement or of reaching the end of the main program does not produce any output to indicate that it is terminating.

1.2.7 Examples of FORTRAN-77 Command Sequences

For a FORTRAN-77 task consisting of:

- The main program MAIN.FTN
- The subroutine SUBR1.FTN
- Several subprograms in the file UTILITY.FTN

you can use the following sequence of commands for compiling, linking, and executing:

```
F77 JOB,JOB= MAIN,SUBR1,UTILITY RET
TKB JOB/FP=JOB,LB:[1,1]F77FCS/LB RET
RUN JOB RET
```

For a more complex task that uses the same FORTRAN-77 source programs but includes the following options:

- A system global common block named PARM
- An increase in the user record-buffer size
- Subroutines in the object module library MATLIB.OLB
- The FORTRAN-77 OTS in library LB:[1,1]F77FCS.OLB
- Array bounds checking in the compiled code

you can use the following sequence of commands:

```
F77 JOB,JOB=MAIN,SUBR1,UTILITY/CK RET
TKB RET
TKB>JOB/FP=JOB,MATLIB/LB,LB:[1,1]F77FCS/LB RET
TKB>/ RET
ENTER OPTIONS:
TKB>COMMON=PARM:RW RET
TKB>MAXBUF=256 RET
TKB>// RET
RUN   JOB RET
```

You can also run this procedure by using indirect command files. For example, suppose the file COMPILE.CMD contains:

```
JOB,JOB=MAIN,SUBR1,UTILITY/CK
```

and the file LINK.CMD contains:

```
JOB/FP=JOB,MATLIB/LB,LB:[1,1]F77FCS/LB
/
COMMON=PARM:RW
MAXBUF=256
//
```

1.3 Using FORTRAN-77 on RSTS/E Systems

This section contains information for the user who wants to compile, task-build, and execute a FORTRAN-77 program on a RSTS/E system. Specifically, it describes how to:

- Invoke the FORTRAN-77 compiler and the RSTS/E Task Builder (with RUN commands or with Concise Command Language (CCL)[1] commands)
- Write RSTS/E file specifications
- Use command
- Use the FORTRAN-77 compiler
- Use the RSTS/E Task Builder
- Execute a task image

1.3.1 RSTS/E File Specifications

For each RSTS/E system program you use, you must specify the input files to be processed and (optionally for the FORTRAN-77 compiler and the Task Builder) the output files to be produced.

The format of a file specification for a RSTS/E system program is as follows:

```
dev:[p,pn]filename.typ
```

[1] Refer to the *PDP-11 FORTRAN-77 Installation Guide* for information on how to install FORTRAN-77 as a CCL command.

dev
The device on which the file is stored or is to be written. You designate the device type by specifying a 2-character device code and, optionally, a unit number. You may also use a logical device name consisting of one to six alphanumeric characters. The device element must be followed by a colon.

[p,pn]
The user account containing the requested file. This account number consists of a project number and a programmer number, each in decimal.

filename
One to six alphanumeric characters. There is no default value for filename.

typ
One to three alphanumeric characters describing the type of data in the file.

You need not explicitly state all the elements of a file specification each time you compile, link, or execute a program. In most cases, when you omit any part of a file specification, a default value is used. Table 1–3 summarizes the applicable default values.

Table 1–3: RSTS/E File Specification Defaults

Optional Element	Default Value	
dev:	SY	
[p,pn]	User's current default PPN (project number, programmer number)	
typ	Depends on usage:	
	Command file	CMD
	Input to the FORTRAN–77 compiler	FTN
	Output from FORTRAN–77 compiler	OBJ
	Source listing from FORTRAN–77 compiler	LST
	Input to Task Builder	OBJ
	Output from Task Builder	TSK
	Map listing from Task Builder	MAP
	Library input to Task Builder	OLB
	Overlay description input to Task Builder	ODL
	Input to executing program	DAT
	Output from executing program	DAT

Refer to the *RSTS/E System User's Guide* for a complete discussion on RSTS/E file specifications.

1.3.2 Command Switches

See Section 1.2.2.

Note that the DCL qualifier /STANDARD=NONE does not work on RSTS/E systems.

1.3.3 Compiling a FORTRAN–77 Program on RSTS/E Systems

The FORTRAN–77 compiler is a system program that produces relocatable object modules from FORTRAN–77 source code.

To invoke the FORTRAN–77 compiler, you type the command line:

```
RUN $F77 RET
```

Or, if the system manager has installed F77 as a CCL command, you can type:

```
F77 RET
```

In either case, after you press the RETURN key, the compiler issues the prompt:

```
F77>
```

You respond to the F77> prompt by entering input and output file specifications (see Table 1-2) as follows:

```
   [obj-file] [,list-file] = infiles-list
```

obj-file

The file specification of the object code file to be created by the compiler. If you do not give a file type in this specification, .OBJ is supplied as a default. This is the default file type expected by the Task Builder when you link the compiled object modules to make an executable file. If you do not want an object file, omit this file specification from the command line.

list-file

The file specification of the listing file created by the compiler. If you do not include a file type in this specification, the compiler supplies .LST as the default. If you do not want a listing file, omit this file specification from the command line. When you include a listing file name, the compiler saves the listing file on disk; you can then print the listing file using the RSTS/E QUE program after the compilation is done. Refer to the *RSTS/E System User's Guide* for a description of the QUE program. The following example shows how to create an object file (OBJECT.OBJ) and a listing file (LISTF1.LST) on disk from an input source file (INPUTF.FTN):

```
F77> OBJECT,LISTF1=INPUTF
```

If you specify a listing file without an object file, you must precede the listing file with a comma to indicate the absence of the object file. For example:

```
F77> ,LISTF1=INPUTF
```

infiles-list

A list of the file specifications of the files that contain the FORTRAN-77 source programs. You can specify more than one input source file in a command line; however, you generally specify only one. When you have multiple specifications, separate them with commas. If you do not provide a file type with this specification, the compiler assumes a default file type of .FTN. For example, to compile three source programs called FILE1, FILE2, and FILE3 into an object module called SINGLE, you enter:

```
F77> SINGLE,SINGLE=FILE1,FILE2,FILE3
```

You can also include the file types, as follows:

```
F77> SINGLE.OBJ,SINGLE.LST=FILE1.FTN,FILE2.FTN,FILE3.FTN
```

You may append to these file specifications any of the compiler command switches listed and described in Section 1.2.4, except the ones noted.

When the compilation is done, the compiler prints another F77> prompt. You can perform as many compilations as you wish before you return to system command level. To exit to the keyboard monitor, type CTRL/Z or CTRL/C.

If F77 has been installed as a CCL command, you can type the entire specification on one line, as follows:

```
   F77 [obj-file] [,list-file] = infiles-list
```

Again, you may include any of the switches listed in Section 1.2.4, except the ones noted.

1.3.4 Task-Building a FORTRAN-77 Program on RSTS/E Systems

The Task Builder is a system program that links relocatable object modules to form an executable task image. The *RSTS/E Task Builder Reference Manual* describes the Task Builder in detail.

1.3.4.1 Using the Task Builder on RSTS/E Systems

You can load the Task Builder into memory by typing a RUN command in the following format:

```
RUN $TKB RET
```

Or, if your system manager has installed TKB as a CCL command, you can type:

```
TKB RET
```

In either case, after you press the RETURN key, the Task Builder prints the TKB> prompt. You then enter a command line to identify the files to be used, as follows:

```
  TKB>[task-file][,map-file] = infiles-list
```

After you press the RETURN key, the Task Builder prints another TKB> prompt. You then:

- Enter additional input files, if any.
- Type a line containing only two slashes(//) to tell the Task Builder to create a task image and to exit with no TKB> prompt.
- Press the RETURN key. (See Section 1.2.5.2 if you are entering any Task Builder options.)

If TKB has been installed as a CCL command, and you want to perform one task-build operation, you can type the whole request on one line, as follows:

```
  TKB [task-file][,map-file] = infiles-list
```

After you press the RETURN key, the Task Builder processes the command line. It then returns you to the keyboard monitor.

The parameters task-file, map-file, and infiles-list use the standard RSTS/E file specification format described in Table 1-2.

The elements in the Task Builder command line are as follows:

task-file

The file specification of the task-image output file created by the Task Builder. If you do not provide a file type in the task-file name, the Task Builder supplies .TSK as a default. Therefore, the following commands are equivalent:

```
TKB FILE1/FP=FILE1

TKB FILE1.TSK/FP=FILE1
```

The task-file specification may be omitted if no task-image file is desired.

map-file

The file specification of the map output file. The map file contains information about the size and location of routines and global symbols within the task image. If you do not provide a file type in the map-file name, the Task Builder supplies .MAP as a default. When you specify a file name, the Task Builder saves the map output on disk. If you do not specify a task-image file specification in the command line, you must precede the map-file name with a comma to indicate the intended absence of the specification. The map-file specification may be omitted if no task-image map file is desired.

infiles-list

The list of input files that contain compiled FORTRAN–77 object modules. You can specify as many input files as can fit in 80 columns in the command line; however, you can place additional input files on additional lines, as long as each specification is contained wholly on one line (not split between or among lines). When you specify multiple object files or libraries, separate them with commas. If you do not give a file type, .OBJ is assumed as a default. For input library files, you must specify the /LB switch following the input file name.

For example, to build a task image for the object-file SINGLE created in Section 1.3.3, when the FORTRAN–77 OTS is included in the system object library (LB:SYSLIB.OLB), you enter:

```
TKB SINGLE/FP,SINGLE=SINGLE
```

Or, if you prefer to include the file types, you enter:

```
TKB SINGLE.TSK/FP,SINGLE.MAP=SINGLE.OBJ
```

Both of these command lines save a copy of the map file (SINGLE.MAP) on disk.

If a separate library contains the FORTRAN-77 OTS routines, you must specify the library name in the Task Builder command line, as shown in the following example:

```
TKB SINGLE/FP,SINGLE=SINGLE,LB:F77FCS/LB
```

If you are using RMS, you must explicitly include a reference to the RMS library in the task-build command line. The previous example would then become:

```
TKB SINGLE/FP,SINGLE=SINGLE,LB:F77RMS/LB,LB:RMSLIB/LB
```

When building a task image with object modules produced by FORTRAN-77, you cannot include in the task object modules from other PDP-11 compilers and OTS routines. Also, you must not combine in a single task object modules created by different PDP-11 compilers.

In addition, a Task Builder command line can contain switches that specify optional file-controlling actions. For example, when you attach the /DA (Debugging Aid) switch to the task image file specification, the Task Builder automatically includes system on-line debugging aid LB:ODT.OBJ in the task image. To negate the /DA switch, you can type either /-DA or /NODA. See Section 1.2.5.1 for the switches that apply to the RSTS/E Task Builder; the RSTS/E Task Builder command switches are also described in the *RSTS/E Task Builder Manual.*

NOTE

> You must include the /FP switch when you build a task. (Refer to Section 5.4.1 for the exception to this rule.) This switch instructs the Task Builder to reserve an area into which the intermediate results of floating-point computations can be placed when job rescheduling occurs. If you omit the /FP switch, you may receive unreliable results.

1.3.4.2 Task Builder Options

See Section 1.2.5.2.

1.3.4.3 Library Usage on RSTS/E Systems

A library can be relocatable or resident. A relocatable library is one that the Task Builder can make a physical part of a task image. A resident library is one that the Task Builder can make a logical part —but not a physical part— of a task image; that is, the Task Builder can link it to the task image but cannot copy it into the task image.

Relocatable Libraries—Relocatable libraries reside in files on disk. From these libraries, the Task Builder copies object modules into the task image of each task that references those modules. You must tell the Task Builder that an input file is contained in a relocatable library by appending the switch /LB to the input file specification of that file. If you do not include a file type with the file name of such a file specification, the Task Builder assumes .OLB as a default. When the Task Builder encounters a library file specification, it includes in the task image being built those modules in the library that contain definitions of any currently undefined global symbols. The system relocatable library and user relocatable libraries are described below.

Resident Libraries—Resident libraries reside in memory, where they are accessed, but not copied, by the tasks that need them. A task may reference one or more resident libraries. You tell the task program to access a resident library by specifying the LIBR or RESLIB option. Section 1.2.5.2 describes these two options.

System Libraries—RSTS/E has a system relocatable library called LB:SYSLIB.OLB and, in addition, has available to it three system resident libraries pertinent to FORTRAN-77.

The Task Builder searches the system relocatable library if any undefined global references are left after it has processed all the input files. If the Task Builder finds the definition of one of these global symbols in the system relocatable library, it includes the appropriate object module in the task.

A system resident library may be available for use with RSTS/E:

RMSRES - A resident library of RMS-11 input/output routines.

Ask your system manager if this library is available to you; your system might not have enough memory to support them.

One or two of the following Task Builder options may link the system libraries to your task:

```
LIBR = RMSRES:RO
```

or

```
TKB> PROG/FP=PROG,MTXLIB/LB,LB:F77RMS/LB,LB:RMSLIB/LB
```

User Libraries—Using the Librarian Utility, you can create your own FORTRAN-77 (or assembly language) relocatable libraries. You then access these libraries by using the /LB switch after the appropriate library name. Refer to the *RSTS/E Programmer's Utilities Manual* for more information on the Librarian Utility.

1.3.5 Executing a FORTRAN-77 Program on RSTS/E Systems

To execute a task, you use a RUN command as follows:

```
RUN filespec
```

filespec
A file specification of the form described in Section 1.3.1.

Generally, you do not need to include all the elements in a file specification. For example, to execute a task file (TASK01.TSK) located in your account on the public disk structure, you type:

```
RUN TASK01.TSK
```

The system assumes SY: as the default device and your account as the default project-programmer number.

1.3.6 Examples of FORTRAN-77 Job Command Sequences

For a FORTRAN-77 task image consisting of:

- The main program MAIN.FTN
- The subroutine SUBRTN.FTN
- Several subprograms in the file SUBPRG.FTN

you can use the following sequence of commands for compiling, task-building, and executing the image:

```
F77 JOB,JOB = MAIN,SUBRTN,SUBPRG [RET]
TKB JOB/FP = JOB,LB:F77RMS/LB,LB:RMSLIB/LB [RET]
RUN JOB [RET]
```

For a more complex task that uses the same FORTRAN-77 source programs but includes the following options:

- A system global common block named PARAM
- An increase in the user record-buffer size
- Subroutines in the object-module library MATLIB.OLB
- The FORTRAN-77 OTS in separate library LB:F77RMS.OLB
- Array bounds-checking in the compiled code

you use a sequence of commands as follows:

```
F77 JOB,JOB=MAIN,SUBRTN,SUBPRG/CK [RET]
```

```
TKB [RET]
TKB>JOB/FP=JOB,MATLIB/LB,LB:F77RMS,LB:RMSLIB/LB [RET]
TKB>/ [RET]
ENTER OPTIONS:
TKB>COMMON=PARAM:RW [RET]
TKB>MAXBUF=256 [RET]
TKB>// [RET]
```

```
RUN JOB [RET]
```

You can also run the above procedure using indirect command files. For example, if the file COMPIL.CMD contains:

```
JOB,JOB=MAIN,SUBRTN,SUBPRG/CK
```

and the file LINK.CMD contains:

```
JOB/FP=JOB,MATLIB/LB,LB:F77RMS,LB:RMSLIB/LB
/
COMMON=PARAM:RW
MAXBUF=256
//
```

then the following sequence is equivalent to the previous example:

```
F77 @COMPIL [RET]
TKB @LINK [RET]
RUN JOB [RET]
```

1.3.7 Programming Considerations for RSTS/E Users

You should note the following programming considerations and restrictions:

- The RSX emulator restricts the use of the memory management (PLAS) directives to resident libraries only; consequently, the use of virtual arrays is not supported.
- RSTS/E does not provide an interface for the set of FORTRAN-77 process-control routines or RSX system directives.
- You cannot extend an existing contiguous file under RSTS/E; you must instead allocate an adequate amount of space when you create a contiguous file under RSTS/E.
- A FORTRAN-77 program must load into no more than 28K words. However, if the RSX emulator support has been added to the system monitor, a program may extend to 31K words. In addition, a program may use up to 32K words if resident libraries are supported.
- The UNITS option for TKB is restricted to the range 1-14 on RSTS/E systems.

NOTE

> You will not receive an error message from the Task Builder if your program exceeds 28K words. However, if your program does surpass the prescribed maximum size, you will receive the run-time error message, "?Illegal byte count for I/O."

- The OTS does not let you supersede an existing file. If you do attempt to create a new file with the same name as that of an existing file, you will receive error number 30: "Open failure."
- A contiguous file cannot be extended on RSTS/E. The initial size of a contiguous file is also the maximum size.
- You can read past EOF records on interactive devices.
- Refer to the *RMS-11 User's Guide* for a list of RSTS/E restrictions on RMS-11.

1.4 Using FORTRAN-77 on VMS Under VAX-11/RSX

This section contains information for the user who wants to compile, task-build, and execute a PDP-11 FORTRAN-77 program on a VMS system.

Specifically, this section describes how to:

- Write VMS file specifications
- Use command switches
- Use the FORTRAN-77 compiler to create an object module
- Use your system's Task Builder to create a task image
- Execute a task image

For more information on using VMS AME, consult the *VAX-11/RSX-11M User's Guide* and the *VAX-11/RSX-11M Programmer's Reference Manual.*

1.4.1 VMS File Specifications

For each VMS system program you use, you must specify the input files to be processed and (optionally for the FORTRAN-77 compiler and your system's Task Builder) the output files to be produced.

The format of a file specification for a VMS system program is as follows:

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

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

[directory]
The named directory containing the desired file.

filename
The file by its name. A filename value can be up to nine characters long.

filetype
The kind of data in the file. A filetype value can be up to three characters long.

version
The version of the file that is desired. Versions are identified by a decimal number, which is incremented by 1 each time a new version of a file is created.

You need not explicitly state all the elements of a file specification each time you compile, task-build, or execute a program. The only part of a file specification that is usually required is the file name. If you omit any other part of the file specification, a default value is used. Table 1–4 summarizes the file specification default values.

Table 1–4: VMS File Specification Defaults

Optional Element	Default Value	
device	User's current default device	
[directory]	User's current default directory	
filetype	Depends on usage:	
	Command file	CMD
	Input to compiler	FTN
	Output from compiler	OBJ
	Input to Task Builder	OBJ
	Output from Task Builder	EXE
	Input to RUN command	EXE
	Compiler source listing	LST
	Task Builder map listing	MAP
	Task Builder library input	OLB
	Task Builder overlay description	ODL
	Input to executing program	DAT
	Output from executing program	DAT
version	Input: highest existing version Output: highest existing version plus 1	

If you request compilation of a source program specified only by a file name, the compiler searches for a file with the specified file name that:

- Is stored on the default device
- Is cataloged under the current default directory
- Has a file type of FTN

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

For example, assume that your default device is DK0, that your default directory is [SMITH], and that you supply the following input or output file specification to the compiler:

```
CIRCLE
```

For input, the compiler searches device DK0 in directory [SMITH] for the highest version of CIRCLE.FTN. For output, the compiler generates the file CIRCLE.OBJ, stores it on device DK0 in directory [SMITH], and assigns it a version that is higher by 1 than any other version of CIRCLE.OBJ currently cataloged in directory [SMITH] on DK0.

1.4.2 Command Switches

Command switches are devices you can use in command lines to specify optional command instructions or inputs: for example, to specify that the compiler compile all lines with a D in column 1.

Command switches are appended to other entities in a command line and have the form:

```
  /switch[:val]
```

switch
A mnemonic that specifies a certain instruction to the compiler or Task Builder.

val
A parameter consisting of an octal or decimal number, or a string of characters.

Many switches have a negative form that negates the action specified by the positive form. You can obtain the negative form generally by following the required slash with a minus sign or the characters NO. For example, /-SP or /NOSP prevents automatic spooling of a program listing.

1.4.3 Compiling a FORTRAN-77 Program

The PDP-11 FORTRAN-77 compiler is a system program that produces relocatable object modules from FORTRAN-77 source code.

You invoke the FORTRAN-77 compiler with the MCR command F77 as follows:

```
   MCR F77 [obj-file] [,list-file] = infiles-list
```

obj-file
The file specification of the object code output file. This file specification may be omitted if no object file is desired. If it is entered, only a file name value is required; a file type value of OBJ is assumed by default if no file type is specified. Therefore, the following commands are equivalent:

```
MCR F77 FILE1=FILE1

MCR F77 FILE1.OBJ=FILE1
```

Note, however, that no listing file is created in either case.

list-file
The file specification of the listing output file. This file specification may be omitted if no listing file is wanted. If it is entered, only a file name value is required; a file type value of LST is assumed by default if no file type is specified. The listing file is saved on disk.

infiles-list
The list of input files that contain the source programs. In many cases, this list contains only one file specification; however, when there is more than one, you must separate the individual specifications with commas. Only a file name is normally required; a file type value of FTN is assumed if no file type is specified.

For example, to compile three source programs called WINKN, BLINKN, and NOD into an object module called SINGLE, you would enter:

```
MCR F77  SINGLE, SINGLE = WINKN, BLINKN, NOD
```

or, if you wish:

```
MCR F77  SINGLE.OBJ,SINGLE.LST=WINKN.FTN,BLINKN.FTN,NOD.FTN
```

In addition, an F77 command line can contain one or more of the compiler switches listed and described in Section 1.2.4.

You can also use the F77 command in interactive mode, which permits you to enter multiple compilation commands (lines). To invoke the interactive mode (if you have installed the image of the FORTRAN-77 compiler as F77), you simply type:

```
MCR F77 RET
```

Regardless of the name under which the PDP-11 FORTRAN-77 compiler is installed, the compiler displays the following prompt:

```
F77>
```

To enter a succession of compilation commands under interactive mode, you type one command line after each prompt, followed by a carriage return, until all commands are entered. Each command line must specify the appropriate input and output files for the program module to be compiled, and any optional switches desired. You then type CTRL/Z. For example, if you want the FORTRAN programs WINKN, BLINKN, and NOD compiled into separate object modules, you can enter a succession of commands as follows:

```
MCR F77 RET
```

From this point on, the compiler issues the F77> prompt.

```
F77>WINKN,WINKN/SP=WINKN RET
F77>BLINKN,BLINKN/SP=BLINKN RET
F77>NOD,NOD/SP=NOD RET
F77> CTRL/Z
```

Note that the compiler types the F77> prompt each time you enter a command, until you type CTRL/Z (^Z) to return system control to MCR.

You can also enter the name of an indirect command file in response to the F77> prompt. For example, if the file COMPILE.CMD contains:

```
WINKN, WINKN/SP=WINKN
BLINKN, BLINKN/SP=BLINKN
NOD, NOD/SP=NOD
```

then the commands

```
F77>@COMPILE RET
F77> CTRL/Z
```

are equivalent to the previous example.

1.4.4 Task-Building a FORTRAN-77 Program

The Task Builder is a system program that links relocatable object modules to form an executable task image. You invoke the Task Builder by entering the MCR command TKB. TKB is described in Section 1.2.5.1.

The object modules to be linked can come from user-specified input files, user libraries, or system libraries. The Task Builder resolves references to symbols defined in one module and referred to in other modules. Should any symbols remain undefined after all user-specified input files are processed, the Task Builder automatically searches the system object library LB:[1,1]SYSLIB.OLB to resolve them.

The default FORTRAN–77 object time system library normally is object library LB:[1,1]F77FCS.OLB, or LB:[1,1]F77RMS.OLB.

Two versions of the OTS I/O support modules for FORTRAN–77 are distributed. One version uses File Control Services (FCS-11), which supports sequential and direct access to sequential files. The other version of the OTS I/O support library uses Record Management Services (RMS–11), which supports sequential, direct, and keyed access to sequential, relative, and indexed files.

The FCS-11 file system is always contained in the system object library (that is, in LB:[1,1]SYSLIB.OLB); the RMS–11 file system is always contained in a separate object library (that is, LB:[1,1]RMSLIB.OLB).

The Task Builder also resolves references to resident common blocks and resident libraries; the task image produced, therefore, is ready to be run under the operating system.

You can also use the Task Builder to build tasks with overlay structures.

1.4.4.1 Using the MCR Command TKB

You use the MCR command TKB to invoke the Task Builder.

The TKB command line has the format:

```
MCR TKB [task-file]/FP[,map-file] = infiles-list
```

task-file

The file specification of the task-image output file. This file specification may be omitted if no task-image file is desired. If a specification is entered, only a file name is required; a filetype value of TSK is assumed if no filetype is specified. Therefore, the commands:

```
MCR TKB  FILE1/FP=FILE1
```

and

```
MCR TKB  FILE1.TSK/FP=FILE1
```

are equivalent. Note, however, that no map file is created in either case.

The following switches may be applied to the task-image file:

/FP

Specifies that the task use the Floating Point Processor (FP11) or floating-point microcode option (KEF11A).

NOTE

> You *must* include the /FP switch when you build a task; if you do not, the task will exit with the FORTRAN run-time message: "TASK INITIALIZATION FAILURE." (Refer to Section 5.4.1 for the one exception to this rule.)

/DA

Specifies that the system debugging aid ODT is to be included in the task.

/MU

Specifies that multiple versions of the task may be run simultaneously. The read-only portions of the task are shared.

map-file

The file specification of the map output file. This file specification may be omitted if no task-image map file is desired. If a specification is entered, only a file name is required; a file type value of MAP is assumed if no file type is specified. The map file is automatically spooled to the line printer. On some operating systems, the map file is automatically deleted after it is printed.

The following switches may be applied to the map file:

/CR

Specifies that a global cross-reference listing is to be appended to the map file.

/SP

Specifies that the map file is to be spooled to the line printer.

infiles-list

The list of input files that contain compiled FORTRAN–77 object modules. (This list may also contain compiled or assembled libraries and modules that were written in a language other than FORTRAN, such as MACRO.) In many cases, this list contains only one file specification; however, when there is more than one specification, you must separate the individual specifications with commas. Only a file name is normally required; a file type value of OBJ is assumed.

The following switches may be applied to input files:

/LB

Specifies that the input file is to be a library file. See Section 1.2.5.3.

/MP

Specifies that the input file is an overlay description file. See Section 1.4.

For example, to build a task image for the object file SINGLE, created in Section 1.4.3, enter the following:

```
MCR TKB SINGLE/FP,SINGLE=SINGLE,LB:[1,1]F77FCS/LB
```

If the default I/O support library on your system is RMS–11, you must explicitly reference RMSLIB in the task-build command line. The previous example then becomes:

```
MCR TKB SINGLE/FP,SINGLE=SINGLE,LB:[1,1]F77RMS/LB,LB:[1,1]RMSLIB/LB
```

You can also use the TKB command in interactive mode, which permits you to enter multiple-line commands. To enter interactive mode, you simply type:

```
MCR TKB RET
```

The Task Builder then displays the following prompt:

```
MCR TKB>
```

You may now enter a single command line that identifies all the input files you want to use to begin the task build, followed by a carriage return. Or you may enter additional input files on as many subsequent lines as you need. When you have entered all your input files, you must type a final line consisting of two slash characters (//), followed by a carriage return (see Section 1.3.4.2 if you are entering any Task Builder options). The double slash signals the Task Builder to begin processing.

1.4.4.2 Task Builder Options

The Task Builder allows numerous options to be specified. Several of these are of particular interest to the FORTRAN-77 user.

To specify options in the MCR command TKB, you must use the Task Builder in interactive mode, and you must terminate command input with a line consisting of a single slash (/) (rather than the double slash described in Section 1.2.5.1). The single slash signals the Task Builder to prompt you, as follows, for option information:

```
ENTER OPTIONS:
TKB>
```

At this point, you can enter as many Task Builder options as you need, one option per line. After you enter each option, the Task Builder automatically prompts you for the next option until you enter a single slash (/) to signal no more options. The Task Builder then proceeds to build the task and to produce any requested output. To exit interactive mode in TKB, enter two slashes (//).

The following Task Builder options can be useful to you as a FORTRAN-77 programmer.

ACTFIL—You can declare the total number of input and output files that a task can open simultaneously, and allocate the proper number of buffers, by entering:

```
ACTFIL = n
```

n
The number, in decimal, of files that can be opened simultaneously and the buffers needed to accommodate them. The default value is 4.

Any attempt to open a file or use a logical unit when space is not available for at least one buffer will cause an error at run time.

The value n includes both explicitly and implicitly opened files.

ASG—You can assign logical unit numbers to physical devices by entering the following.:

```
ASG = dev1:n1:n2:...,dev2:m1:m2:...
```

dev
A physical device name.

n
A valid logical unit number.

m
A valid logical unit number.

The default device assignments are as follows:

```
ASG = SY0:1:2:3:4,TI0:5,CL0:6
```

EXTTSK—You can allocate additional buffer space for RMS-11 input/output by using the option

```
EXTTSK = n
```

n
The number, in decimal, of words to allocate. The value assigned by this option may be overridden by the /INC option on the RUN command (see Section 1.2.6).

For information on how to determine the amount of buffer space a program may need, refer to the *RMS-11 MACRO-11 Reference Manual.*

The EXTTSK option is more efficient than the ACTFIL option because:

- The amount of space can be more accurately specified.
- The space allocated by EXTTSK does not require disk space in the task-image file.

When you use an operating system that supports the Extend Task system directive, the RMS-11 version of the OTS attempts to extend the buffer space dynamically.

FMTBUF—The default size of the buffer used to contain the internally compiled form of a format specification stored in an array is 64 bytes. You can increase the size of this buffer by entering:

```
FMTBUF = n
```

n
The decimal size, in bytes, of the run-time format compilation buffer.

The total size needed for format compilation is equal to the largest run-time format specification used by the program. For information on how to determine the amount of space needed to store a given format, refer to the *PDP-11 FORTRAN-77 Object Time System Reference Manual.*

GBLPAT—To patch FORTRAN logical unit 0 to a valid system logical unit, use the option

```
   GBLPAT= main-prog:$LUNO:n
```

main-prog
The name of your main program segment.

n
A system logical unit number in the range 1 through 99 (see Section 2.1.3).

MAXBUF—The default maximum record size for input/output is set at 133 (decimal) bytes. You can increase this record size by entering:

```
MAXBUF = n
```

n
The number of bytes (in decimal).

The default generally is adequate for sequential input/output. If sequential, direct, or keyed access operations are performed with records larger than 133 bytes, you must use this option, as follows, to specify the size of the largest record you intend to process.

For formatted records:

```
MAXBUF = RECL
```

For unformatted records:

```
MAXBUF = 4*RECL
```

For segmented records (see Section 2.2.3.3 for a definition of segmented records):

```
MAXBUF = (4*RECL)+2
```

The two extra bytes for segmented records are the segment control bytes (see Section 2.2.3.3).

UNITS—The default number of logical units available to a program is 6 (logical units 1 through 6, inclusive). You can set this number smaller or larger at task-build time by entering:

```
UNITS = n
```

n
The number, in decimal, of logical units desired, from 0 to 99.

However, you should be aware that increasing the number of default units available will increase task size.

The default device and file name associated with a logical-unit number are discussed in Section 2.1.1.

When you need to assign devices to the units you have specified with the UNITS option, use the ASG option discussed earlier in this section. If you need more units than the six provided as the default, you must enter the UNITS option before you make any assignments with ASG.

1.4.4.3 Library Usage on VMS Systems

There is only one type of VMS library: relocatable. A relocatable library is a collection of object modules that the Task Builder can make a physical part of a task image.

Relocatable Libraries—Relocatable libraries are stored in files on disk. From these libraries, the Task Builder copies object modules into the task image of each task that references those modules. You must tell the Task Builder that an input file is contained in a relocatable library by attaching the switch /LB to the input file specification of the file. If you do not include an extension with the file name of such a specification, the Task Builder assumes .OLB as a default. When the Task Builder encounters a library specification, it includes in the task image being built those modules in the specified library that contain definitions of any currently undefined global symbols.

System Libraries—Each VMS system has a system relocatable library, which follows:

```
LB:[1,1]SYSLIB.OLB
```

The Task Builder automatically searches the system relocatable library to see if any undefined global references remain after all the input files have been processed. If the definition of one of these undefined global symbols is found, the appropriate object module is included in the task being built.

User Libraries—Using the Librarian Utility, you can construct your own FORTRAN-77 (or assembly language) relocatable libraries. You then access these libraries by using the appropriate library switch, as described in preceding sections. Consult the *VAX-11/RSX-11M User's Guide* for further information on the Librarian Utility.

For example, if MATRIXLIB.OLB is a relocatable library containing matrix manipulation routines and PROG is the object file of a compiled FORTRAN-77 program that calls the matrix routines, you could enter the following command line for the Task Builder:

```
MCR TKB PROG/FP=PROG,MATRIXLIB/LB
```

1.4.5 Executing a FORTRAN-77 Program

To begin task execution once you have built a task image, you enter a RUN command of the form:

```
  RUN filespec
```

filespec
The file specification of the file containing the task image.

You can end a task before its normal completion by typing CTRL/C (^C).

You should not suspend task execution with a PAUSE statement under VMS. There is no way to resume execution once the task has paused.

In batch mode, the PAUSE statement types the display to the log file, but the program does not pause.

A task that terminates as a result of a CALL EXIT statement or of reaching the end of the main program does not produce any output to indicate that it is terminating.

1.4.6 Examples of FORTRAN–77 Command Sequences

For a FORTRAN–77 task consisting of:

- The main program MAIN.FTN
- The subroutine SUBR1.FTN
- Several subprograms in the file UTILITY.FTN

you can use the following sequence of commands for compiling, linking, and executing:

```
MCR F77 JOB,JOB= MAIN,SUBR1,UTILITY RET
MCR TKB JOB/FP=JOB,LB:[1,1]F77FCS/LB RET
RUN JOB RET
```

For a more complex task that uses the same FORTRAN–77 source programs but includes the following options:

- An increase in the user record-buffer size
- Subroutines in the object module library MATLIB.OLB
- The FORTRAN–77 OTS in separate library LB:[1,1]F77FCS.OLB or LB:[1,1]F77RMS.OLB
- Array bounds checking in the compiled code

you can use the following sequence of commands:

```
MCR F77 JOB,JOB=MAIN,SUBR1,UTILITY/CK RET
MCR TKB RET
TKB>JOB/FP=JOB,MATLIB/LB,LB:[1,1]F77FCS/LB RET
TKB>/ RET
ENTER OPTIONS:
TKB>MAXBUF=256 RET
TKB>// RET
RUN   JOB RET
```

You can also run this procedure by using indirect command files. For example, suppose the file COMPILE.CMD contains:

```
JOB,JOB=MAIN,SUBR1,UTILITY/CK
```

and the file LINK.CMD contains:

```
JOB/FP=JOB,MATLIB/LB,LB:[1,1]F77FCS/LB
/
COMMON=PARM:RW
MAXBUF=256
//
```

The following is now equivalent to the previous example:

```
MCR F77 @COMPILE RET
MCR TKB @LINK RET
RUN JOB RET
```

1.5 Overlays

The overlay facility provided by the Task Builder allows large programs to be executed in relatively small areas of main memory. An overlaid program is essentially a program that has been broken down into parts, or overlays, that are loaded into memory automatically during program execution.

You construct an overlaid program by providing a single file as input to the Task Builder. This file describes the structure of the overlaid program and the actual input files and libraries. You indicate an overlay file in TKB commands with the /MP qualifier on a single input file. For example:

```
TKB A/FP = A/MP
```

No other input files need be specified. The default file type for an overlay description file is ODL.

To specify the structure of an overlay, you use the Overlay Description Language (ODL).

The following sections provide an introduction to the Task Builder Overlay Description Language (ODL) and information about building simple overlaid FORTRAN-77 programs. Consult your operating system's Task Builder manual for more detailed information about overlays and building overlaid programs; also see Section 2.6.5 for information on task-building programs with RMS-11 using overlays.

1.5.1 Introduction to the Overlay Description Language

You can build overlay structures using three ODL statements:

.ROOT specifies the tree structure of an overlay

.FCTR specifies a single branch of an overlay tree, called a factor or segment

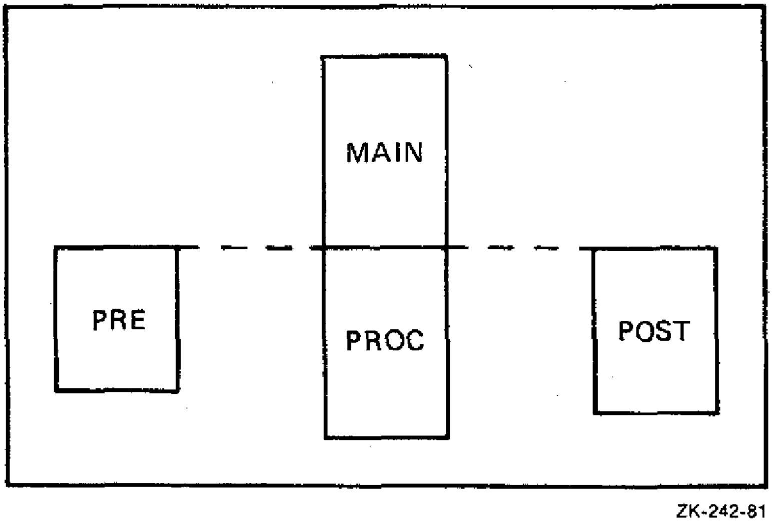
.END indicates the end of an overlay description

For example, suppose a FORTRAN-77 program consists of a main program (MAIN.OBJ) that performs input and output and calls three subroutines: One subroutine does preprocessing of the data (PRE.OBJ); one subroutine does the main processing function of the program (PROC.OBJ); and one subroutine does postprocessing of the data (POST.OBJ). The following ODL statements specify an overlay structure having a resident portion that consists of the main program and three overlays that share the same memory locations. Each overlay contains a single subroutine. Figure 1-2 illustrates this overlay structure. The ODL statements to create this structure are as follows:

```
        .ROOT  MAIN-*(A,B,C)
A:      .FCTR  PRE
B:      .FCTR  PROC
C:      .FCTR  POST
        .END
```

In this example, the .ROOT statement declares the tree structure; the .END statement indicates the end of the ODL statements; and the names A, B, and C specify object modules, libraries, other overlay segment factor names, or indirect ODL file names (if they are preceded by an (@) symbol). Commas separate descriptions of overlay segments that occupy the same memory location; parentheses serve to group these descriptions. Dashes separate descriptions of modules that are concatenated into a single segment. The asterisk indicates that the overlay segments are to be loaded automatically whenever a call is made to a subprogram in the overlay segment.

Figure 1–2: Simple Overlay Structure



A path in an overlay structure is any route from the root of the structure that follows a series of branches to an outermost segment of the tree. Figure 1–2 shows only three short paths: MAIN-PRE, MAIN-PROC, and MAIN-POST. A program in one overlay segment may call a subprogram in another segment only when the two segments occur on a common path. For example, MAIN may call PRE, PROC, or POST; however, the three subroutines cannot call each other.

Figure 1–3 shows a more complex structure specified by the following ODL statements:

```
        .ROOT  A-B-*(C,FCTR1)
FCTR1:  .FCTR  D-*(E,F,G)
        .END
```

The paths in this structure are A-B-C, A-B-D-E, A-B-D-F, and A-B-D-G.

1.5.2 Building Overlaid FORTRAN–77 Programs

When building overlaid FORTRAN–77 programs, you should pay special attention to the following:

- Specifying the FORTRAN–77 OTS library
- Declaring common blocks
- Declaring the associated variable in a DEFINEFILE or OPEN statement
- Specifying the RMS–11 library (if used)

If the FORTRAN–77 OTS is in the default system library, no additional specification is necessary. If the FORTRAN–77 OTS is a separate library, and FCS-11 is used, then each segment or branch of the overlay structure must explicitly refer to the FORTRAN–77 OTS library as the last file specified. On the other hand, if the FORTRAN–77 OTS is a separate library, and RMS–11 is used, then each segment or branch of the overlay structure must explicitly refer to the FORTRAN–77 OTS library as the next-to-last file specified, with the RMS–11 library specified as the last file. For example, the ODL file for the example in Figure 1–3 must be written as follows:

For FCS-11

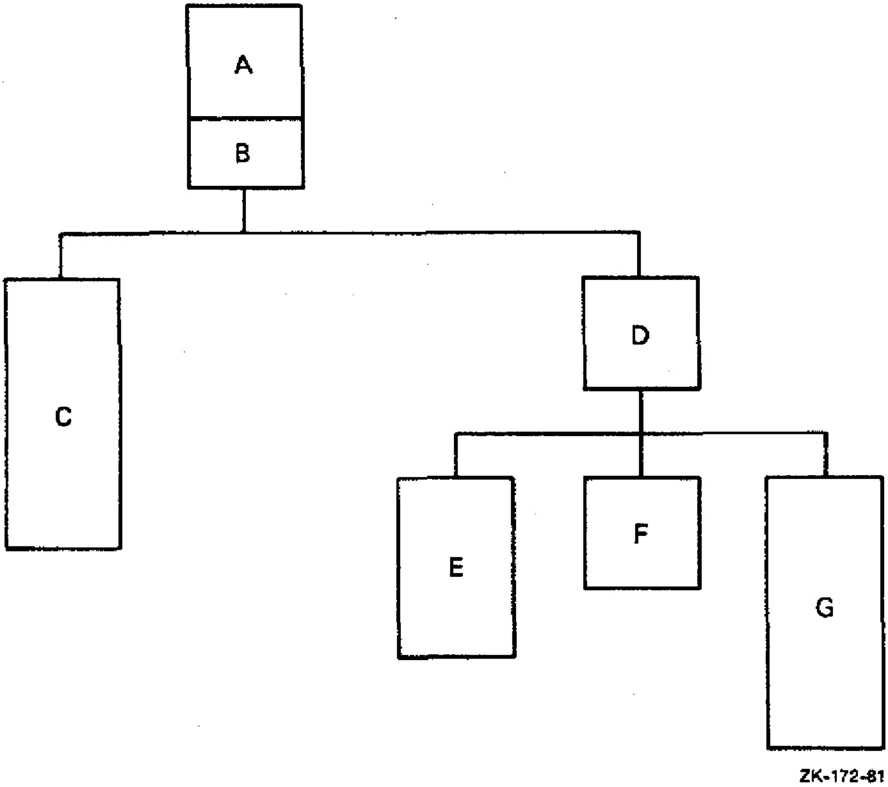
```
         .ROOT  A-B-L-*(C-L,FCTR1)
FCTR1:   .FCTR  D-L-*(E-L,F-L,G-L)
L:       .FCTR  LB:F77FCS/LB
         .END
```

For RMS–11

```
         .ROOT  A-B-L-R-*(C-L-R,FCTR1)
FCTR1:   .FCTR  D-L-R-*(C-L-R,F-L-R,G-L-R)
L:       .FCTR  LB:F77RMS/LB
R:       .FCTR  LB:RMSLIB/LB
         .END
```

If your program refers to user libraries, these libraries must be explicitly referenced by each overlay segment that needs them.

Figure 1–3: Overlay Structure



FORTRAN–77 common blocks are allocated on each overlay path in the lowest overlay segment in which they are referenced. Therefore, when a new overlay path is loaded, the data in the common blocks is lost. If separate overlay paths are to share common data, the common blocks containing this data must be either declared in the root segment of the overlay or specified in a SAVE statement. If the data is declared common only in the overlay segments, separate common areas for each segment are established and the data is not shared.

For example, suppose the subroutines shown in Figure 1-2 (PRE, PROC, POST) communicate using common blocks. If the same common blocks are not declared common in MAIN, three independent common areas with the same name will be established, one each for PRE, PROC, and POST. When PROC overlays PRE, the data in the common block(s) of PRE will be lost. In general, when one segment overlays another, data unique to the overlaid segment is lost.

If you use the SAVE statement to protect common data items, you should be aware that the SAVE statement causes the size of the root segment of an overlay—and therefore the task size—to become larger. This enlargement occurs because using the SAVE statement has the effect of pulling into the root segment of an overlay the $SAVE PSECT and the PSECTs of any named common blocks mentioned in the SAVE statement. (The blank common block PSECT (.$$$$), if present, is pulled into the root segment whether or not a SAVE statement is used, except when the /NOF77 switch is set; under /NOF77, .$$$$ is never pulled into a root segment.) The $SAVE PSECT contains the variables and array elements mentioned in a SAVE statement.

The SAVE statement requires Task Builder support to run an overlaid FORTRAN-77 program in which subprograms that access saved variables reside in different segments of the overlay. Task Builder support is provided beginning with Version 4.0 of RSX-11M, Version 2.0 of RSX-11M /M-PLUS, and Version 7.2 of RSTS/E. If you are not running a supported operating system and are running an overlaid program, you can assure access to saved variables as follows: Place variables or COMMON statements that contain saved variables in the root segment of the overlay. The value of saved variables is retained between subprogram calls.

The associated variable in any DEFINEFILE or OPEN statement must be declared in a common block that is allocated in the root segment.

You can overlay a FORTRAN-77 program in one of three ways:

- You can overlay only the program
- You can overlay only the FORTRAN-77 OTS (and RMS-11, if used)
- You can overlay both the program and the FORTRAN-77 OTS (and RMS-11, if used)

Section 2.6.5 provides information about the RMS-11 overlays used by the RMS-11 version of the FORTRAN-77 OTS. Section 5.4.8 describes the OTS overlay files that are available.

The *FORTRAN-77 Object Time System Reference Manual* describes overlaying the FORTRAN-77 OTS modules in more detail.

1.6 Debugging a FORTRAN-77 Program

FORTRAN-77 provides several aids for finding and reporting errors:

- DEBUG lines in source programs

  FORTRAN-77 statements containing a "D" in column 1 can be added for debugging purposes. During program development, you can use these statements and the /DE switch to type out intermediate values and results. After the program runs correctly, you can treat these statements as comments by recompiling without the /DE switch.

- Traceback facility

  The compiled code and the OTS provide information on the program unit and line number of a run-time error. A list, following the error message, shows the sequence of calling program units and line numbers. The amount of information provided in the list is determined by the /TR switch during compilation. See Section C.3 for the exact format and content of the traceback.

- The debugging program ODT, a user-interactive debugging aid

  You include ODT in a task by specifying the /DA switch on the task image file specification during task building. When using ODT, you should have the machine language code listing of the program (specify the /LI:3 compiler switch) and the task-build map. See the *IAS/RSX-11 ODT Reference Manual* for further information.

- PDP-11 FORTRAN-77 Symbolic Debugger

  If your site has installed the PDP-11 FORTRAN-77 symbolic debugger, you can use its facilities to provide a more thorough debugging than any of the above. The symbolic debugger is interactive and can refer to program locations symbolically and give symbolic output. With the debugger, you can control program execution in a variety of ways: You can set breakpoints and tracepoints; step through your program by line or instruction; and step into or over called routines. You can examine or deposit data in a variety of formats. For complete information, see the *PDP-11 FORTRAN-77 User's Guide* or the *PDP-11 FORTRAN-77 Symbolic Debugger User's Guide.*

Chapter 2

FORTRAN–77 Input/Output

This chapter describes input/output (I/O) as implemented in PDP–11 FORTRAN–77. In particular, it provides information about FORTRAN–77 I/O in relation to the two supporting I/O subsystems: File Control Services (FCS-11) and Record Management Services (RMS–11).

2.1 FORTRAN–77 I/O Conventions

Certain conventions for logical device and file name assignments, and for implied logical units, are common to I/O operations involving either of the I/O subsystems mentioned above.

2.1.1 Device and File Name Conventions

FORTRAN logical unit numbers correspond one-to-one with the operating system's logical units (except FORTRAN logical unit 0, which must be mapped to a system logical unit number other than 0; see Section 2.1.3). Default device assignments are made by the Task Builder for each logical unit allocated for a task.

*Listed in Table 2–1 are the default logical device and file name assign*ments. You can change default device assignments at the following times: (1) prior to execution, by using the appropriate operating system command; (2) at task-build time, by using the Task Builder ASG option (see Section 1.2.5.2); (3) at execution time, by using the ASSIGN system subroutine (see Section D.2) or an OPEN statement.

The default file name conventions hold for logical units not listed below; for example, unit number 12 has a default file name of FOR012.DAT. The default device assignment for logical units not listed is the system disk, SY:.

You may use any combination of valid logical unit numbers; however, there is an imposed maximum number of units that can be active simultaneously. This number depends on the number of buffers allocated and the number of buffers required for each logical unit (usually 1).

Logical unit numbers are allocated consecutively. Therefore, for example, even though only logical units 3 and 17 are being used, units 1 through 17 must be allocated.

When a logical unit is closed, the default file name assignment that existed at the start of task execution is reestablished; the default device assignment becomes undefined.

Table 2–1: FORTRAN–77 Default Logical Device Assignments

Logical Unit Number	Default Device	Default File Name
0	(Mapped to a system logical unit other than 0)	
1	System disk, SY:	FOR001.DAT
2	System disk, SY:	FOR002.DAT
3	System disk, SY:	FOR003.DAT
4	System disk, SY:	FOR004.DAT
5	User's terminal, TI: or TT:	FOR005.DAT
6	System listing unit, CL:	FOR006.DAT
.	.	.
.	.	.
14	(RSTS/E limit)	FOR014.DAT
.	.	.
.	.	.
99	System disk, SY:	FOR099.DAT

NOTE

The device assignment to a logical unit is not affected by a CLOSE operation. However, this convention is subject to change in future releases and should not be relied on. If the device assignment of a unit is changed by a CALL ASSIGN or an OPEN statement, it is recommended that all CALL ASSIGN or OPEN statements referencing that unit explicitly specify the device to be used.

2.1.2 Implied-Unit Number Conventions

Certain I/O statements do not require explicit logical unit specifications. These statements, and their equivalent forms, are listed in Table 2–2.

From Table 2–2, you can see that a formatted READ statement of the form:

```
READ f,list
```

is equivalent to:

```
READ(1,f)list
```

In a program, these two forms function identically. If logical unit number 1 is assigned to a terminal, input comes from this terminal no matter which of the above READ formats you use.

The PRINT, ACCEPT, and TYPE statements implicitly refer to logical units 6, 5, and 5, respectively.

Table 2–2: Implied Unit Numbers

Statement Type	Equivalent Form		
READ	f, list	READ	(1,f) list
PRINT	f, list	WRITE	(6,f) list
ACCEPT	f, list	READ	(5,f) list
TYPE	f, list	WRITE	(5,f) list

2.1.3 Mapping FORTRAN–77 Logical Unit 0 to a System Unit

The default mapping of FORTRAN–77 logical unit 0 is to system logical unit 0; however, 0 is not a valid system logical unit number. Therefore, to map FORTRAN–77 logical unit 0 to a valid system logical unit, use the

GBLPAT option (Section 1.4.4.2) when task-building your program, as follows:

```
>TKB
TKB> PROG = PROG,LB:[1,1]F77FCS/LB
TKB> /
TKB> ENTER OPTIONS:
TKB> GBLPAT = PROG:$LUNO:n
TKB> //
```

where n is a valid system logical unit number.

This command sequence patches global symbol $LUN0 in program segment PROG to system logical unit number n.

2.2 Files and Records

This section discusses file structures, record access modes, and record formats in the context of the capabilities of the FCS and RMS I/O subsystems.

2.2.1 File Structure

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

The term "file organization" refers to the way records are arranged within a file; the term "record access" refers to the method by which records are read from a file or written to a file. A file's organization is specified when the file is created, and cannot be changed. Record access is specified each time a file is opened, and can be different each time the same file is opened. This section discusses file organization; Section 2.2.2 discusses record access. Table 2–3 shows the valid record access modes for each file organization.

Through its two I/O subsystems, FORTRAN–77 supports three file organizations: sequential, relative, and indexed. Table 2–3 summarizes which file organizations are available to the various I/O subsystems.

Table 2–3: **Availability of File Organization**

	FCS-11	RMS-11	RMS-11K
Sequential	X	X	X
Relative	–	X	X
Indexed	–	–	X

The organization keyword in the OPEN statement specifies the organization of a file, as described in Section 2.3.7.

2.2.1.1 Sequential Organization

A sequential organization file, or sequential file, consists of records arranged in a physical sequence that is typically identical to the order in which the records are written to the file: the first record in the file is the first record written, the second record in the file is the second record written, and so forth.

Sequential file organization is permitted on all devices supported by the FORTRAN–77 system, and is supported by the FCS-11 and RMS–11 I/O subsystems.

The sequential files created under the FCS-11 subsystem are compatible, both structurally and functionally, with sequential files created under the RMS subsystem. Therefore, you can freely interchange sequential files among all FORTRAN–77 programs.

2.2.1.2 Relative Organization

A relative organization file, or relative file, consists of a series of numbered positions, called cells, that can either contain a single record or remain empty. These cells are of fixed, equal length and are numbered consecutively from 1 to n, where 1 is the first cell and n is the last cell.

Relative organization allows you to place a record in a file at any position relative to the beginning of the file. As a result, you can retrieve a record simply by specifying that record's relative record number. Conceptually, then, a relative file is similar to a sequential file processed under direct access (see Section 2.2.2.2). The one important distinction is that you can delete a record from a relative file (simply by specifying the appropriate relative record number).

Once a record has been deleted from a relative file, the cell containing it is no longer a logical part of the file, and any attempt to direct-access that cell produces error message #36: "ATTEMPT TO ACCESS NON-EXISTENT RECORD."

Relative files can be stored only on disk and are supported only by the RMS I/O subsystems.

2.2.1.3 Indexed Organization

An indexed organization file, or indexed file, consists of records that are arranged logically according to the value of an alphanumeric or integer field (called a key field) contained in each record. Unlike the records in a sequential or relative file, the records in an indexed file are not necessarily stored contiguously, but may be widely dispersed on disk.

When you create an indexed file, you must designate a specific field, common to each record in the file, as a primary key. The value of this field in any one record determines the position of this record in a file.

You can designate additional fields in the records of an indexed file as alternate keys. These fields do not affect the placement of records in the file (unless the file was created to allow duplicate primary keys, in which case the records actually having duplicate primary keys are ordered by an alternate key). However, each alternate key, like the primary key, provides a way to locate a record within a file. You can specify up to 255 keys for an indexed file using an appropriate RMS utility (see the *RMS-11 User's Guide*). You can also specify keys with an OPEN statement; however, the maximum number you can specify with an OPEN statement depends on the total number of parameters you have specified in the OPEN statement.

Regardless of the means by which they are created, you can access, with indexed READ statements, up to 255 keys from a FORTRAN-77 program.

An indexed file contains a tree-structured table, called an index, for each designated key field. Each entry in an index is a pointer to a set of records, called a bucket, located at the base of the tree. The bucket contains the record with the designated key value and zero or more records with lower key values (or the same key values if the key is an alternate key). A bucket is a unit of I/O transfer consisting of a fixed number of bytes specified by the BLOCKSIZE keyword (see Section 2.3.2).

Both the number of key fields and the size of the bucket are established when you create a file; you cannot change these parameters with subsequent OPEN statements. When you add or modify records, RMS automatically updates the indexes and creates additional entries as needed.

Indexed files permit the most flexible record access. This flexibility is facilitated by the fact that you can use any field in a record as a key and can have multiple keys.

When a FORTRAN–77 program creates an indexed file, the primary key of the records of that file are restricted in two respects: (1) duplicate primary keys are not allowed because the value of each primary key must be unique; and (2) when a record in the file is rewritten, its primary key cannot be changed. These restrictions do not apply to alternate keys. When an indexed file is created by a means other than a FORTRAN–77 program, and in such a way as to support changes to, and duplicates of, primary keys, that file may subsequently be used by a FORTRAN–77 program even though there are duplicate primary keys and the values of any of the primary keys can be changed by the program.

Indexed files can be stored only on disk and are available only if RMS–11 is available on your system.

2.2.2 Access to Records

You can select records for processing by the following methods:

- Sequential (including append) access
- Direct access
- Keyed access

Table 2–4 summarizes the ways in which each of the three file organizations can be accessed.

Table 2–4: Access Modes Per File Organization

	Access			
Organization	Sequential	Direct	Keyed	Append
Sequential	X	X[1]		X[2]
Relative	X	X		
Indexed	X		X	

[1] Fixed-length records only.

[2] Append access to a sequential file consists of opening the file for sequential access and initially positioning the current record printer at the end of the file.

The FCS-11 I/O subsystem supports only sequential and direct access (to sequential files); keyed access is supported only by RMS-11 software.

2.2.2.1 Sequential Access

Sequential access means, as the term implies, that records are processed in sequence. The exact nature of this processing sequence depends on the organization of the file. For sequential files, the processing sequence consists of the physical progression of the records in the file, from first created to last created. Processing a sequential file under sequential access requires that a desired record be read only after all records physically preceding it have been read, and that a new record be written only to the current end of the file. For relative files, the processing sequence consists of the numerical order of the record cells (some of which may not have a record in them). Reading a relative file under sequential access requires that a desired record be read only after all existing records preceding it have been read (empty cells are passed over). Writing to a relative file under sequential access allows a new record to be written at any point. For example, if records 1 and 2 are read (in sequential access mode) in a relative file consisting of 24 record cells, and then a record is written, the new record is written into cell 3 of the file, replacing any old record that may have been there. (The concept of writing a record into a cell already containing a record is a FORTRAN–77 concept.)

The processing sequence for an indexed file consists of an index of ascending key values; a corresponding physical sequence may or may not exist. Reading an indexed file under sequential access requires that only the desired record be read. New records may be added at any point, with the key values within a record determining the record's position.

2.2.2.2 Direct Access

Direct access means that records are selected for processing on the basis of their position relative to the beginning of a relative or sequential file. Only one record needs to be read and a new record can be added at any point. Each READ or WRITE statement must include a relative record number that specifies the record to be read or written.

You can direct-access relative files and sequential files containing fixed-length records that reside on disk, but you cannot direct-access indexed files. Because direct access uses cell numbers to identify and find records, you can issue successive READ or WRITE statements requesting records that either precede or follow previously requested records.

For example, the statements

```
READ(UNIT=12,REC=24)XARRAY
READ(UNIT=12,REC=20)ZARRAY
```

transfer the data in record 24 of the file connected to logical unit 12 to the variable XARRAY, and the data in record 20 of the same file to the variable ZARRAY.

Using direct access to read records in an RMS-11 sequential or relative file may result in FORTRAN run-time error 36 if the specified record was never written. FORTRAN run-time error 36 may also occur if the specified record of a relative file has been deleted.

2.2.2.3 Keyed Access

Keyed access means that records are selected for processing on the basis of alphanumeric strings or integer values, called keys, that identify the desired records. Each indexed READ statement contains a key value that is used to locate the record to be read. The key value is compared against index entries until the bucket containing the record is located. The bucket is then read until the exact record is located.

To insert a new record in an indexed file, you specify in the I/O list of an indexed WRITE statement an item that has previously been defined as a key for the records in the relevant file; you do not specify a KEY= value in the WRITE statement. For example, if NAME has previously been defined in an OPEN statement as a key for the records of an indexed file, to insert a new record in that file you can use the following statement:

```
WRITE (UNIT=10, ERR=9999)ORDER, NAME,
1 ADDRESS, CITY, STATE, ZIP, ITEM
```

Keyed access is valid only for indexed files.

See Chapter 7 for more information on using indexed files.

2.2.3 Record Formats

Records can be stored in a file in one of three formats:

- Fixed length
- Variable length
- Segmented (sequential files only)

The format that applies in a particular case depends on the organization of the file.

NOTE

The term "record format" refers to whether a record is fixed length, variable length, or segmented; the term "record type" refers to whether a record is formatted or unformatted. "Record type" should not be confused with the keyword RECORDTYPE.

2.2.3.1 Fixed-Length Records

When you specify fixed-length records for a file (see Section 2.3.10), you are specifying that all records in the file are to contain the same number of bytes; you specify the size of these records by means of the RECORDSIZE keyword of the OPEN statement (see Section 2.3.9). If the record numbers are to be computed correctly, a sequential file to be opened for direct access must contain fixed-length records.

You can use fixed-length records with sequential, relative, or indexed files.

Each fixed-length record in a relative file contains an extra byte, called the deleted-record control byte. The record overhead in an indexed file consisting of fixed-length records is seven bytes per record.

Fixed-length records always start on a word boundary. An extra byte, called the "pad byte," is allocated if the record length of a fixed-length record is odd.

2.2.3.2 Variable-Length Records

Variable-length records can contain any number of bytes, up to a specified maximum. This maximum can be specified by the RECORDSIZE keyword of the OPEN statement (see Section 2.3.9).

You can use variable-length records with sequential, relative, or indexed file organizations.

Each variable-length record is prefixed by a count field that indicates the number of data bytes in the record. The count field comprises two bytes on a disk device and four bytes on magnetic tape.

Variable-length records in relative files are actually stored in fixed-length cells, the size of which must be specified by the RECORDSIZE keyword of the OPEN statement (see Section 2.3.9). The count field in variable-length records in a relative file specifies the largest record that can be stored in that file. Each variable-length record in a relative file contains three extra bytes, two for the count field and one for deleted-record control. Each variable-length record in an indexed file contains nine extra bytes.

You can make the count field of a variable-length record available to a program by means of a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes of data should be read by an I/O list.

2.2.3.3 Segmented Records

A segmented record is a single logical record consisting of one or more variable-length records (segments). The length of a segmented record is arbitrary; however, the length of the segments themselves is specified by the RECORDSIZE keyword (see Section 2.3.9). Segmented records are useful when you want to write exceptionally long records. Unformatted sequential records written to sequential files are, by default, stored as segmented records.

The segmented record is unique to FORTRAN and can be used only with unformatted sequential files under sequential access. You should not use segmented records for files that will be read by programs written in languages other than FORTRAN.

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

- The first segment in the segmented record

- The last segment in the segmented record
- The only segment in the segmented record
- A segment in the segmented record other than one of the above

This control information is contained in the first two bytes of each segment of a segmented record.

When you wish to access an unformatted sequential file that contains fixed-length or variable-length records you must specify RECORDTYPE='FIXED' or RECORDTYPE='VARIABLE' when you open the file; otherwise, the first two bytes of each record will be misinterpreted as control information, and errors will probably result.

2.3 OPEN Statement Keywords

The following sections supplement the OPEN statement description that appears in the *PDP-11 FORTRAN-77 Language Reference Manual*. In particular, implementation-dependent and/or system-dependent aspects of certain OPEN statement keywords are described. This section does not discuss all the keywords that apply to the OPEN statement.

2.3.1 BLANK

BLANK in an OPEN statement controls the interpretation of blanks in numeric input fields. The default is BLANK='NULL' (blanks in numeric input fields are ignored).

If a logical unit is opened by means other than an OPEN statement, a default equivalent to BLANK='ZERO' is assumed (that is, blanks in numeric input fields are treated as zeros).

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 for these descriptors, embedded and trailing blanks are ignored; the value affected 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 /F77 switch determines whether a default of BLANK='NULL' or BLANK='ZERO' is assumed, as illustrated below:

```
    OPEN(UNIT=1, STATUS='OLD')
    READ(1,10)I,J
10  FORMAT(2I5)
    END

       Data record:   1 2    12

    Assigned values:

        /F77           /NOF77
        I=  12        I=  1020
        J=  12        J=    12
```

If your program treats blanks in numeric input fields as zeros, and you do not want to use the /NOF77 switch, include BLANK='ZERO' in the OPEN statement or use the BZ edit descriptor in the FORMAT statement.

2.3.2 BLOCKSIZE

BLOCKSIZE specifies the physical I/O transfer size for a file. A BLOCKSIZE specification has the form:

```
BLOCKSIZE = bks
```

For magnetic tape files, the value of bks specifies the physical block size in the range 18 to 32767 bytes. The default value is 512 bytes.

For tape files created through the RMS-11 subsystem, the maximum block size is 8192.

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

For relative and indexed files, the value of bks is rounded up to an integral number of 512-byte blocks and used to specify the RMS-11 bucket size, in the range 1 to 32 blocks (1 to 15 on RSTS/E). The default value is the smallest value capable of holding a single record.

When you select a blocksize, and thereby determine the bucket size used by RMS-11, you should consider the performance effects of the following factors: file organization, record format, record size, and the internal information that RMS-11 maintains in each bucket. For example, a large bucket size generally speeds up sequential access to a file because fewer

I/O transfers are required. On the other hand, a minimal bucket size means that minimal I/O buffer space is required.

Consult the *RMS-11 User's Guide* for information on determining optimal bucket size.

2.3.3 BUFFERCOUNT

BUFFERCOUNT specifies the number of memory buffers. A BUFFERCOUNT specification has the form:

```
BUFFERCOUNT = bc
```

The range of values for bc is -1 through 255; a buffercount of -1 specifies that a unit will be opened in block mode rather than record mode. The size of each buffer is determined by the BLOCKSIZE keyword. Therefore, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144.

The default value is one buffer for each sequential or relative file, and two buffers for each indexed file.

2.3.4 DISPOSE

DISPOSE specifies the disposition of a file at the time the file is closed. A DISPOSE specification has the form:

```
            'SAVE'
DISPOSE=    'KEEP'
DISP=       'DELETE'
            'PRINT'
```

DISPOSE cannot be used to save or print a scratch file, or to delete or print a read-only file. A DISPOSE parameter in a CLOSE statement always supersedes a disposition specified in an OPEN statement.

On an IAS operating system, a file printed under DISPOSE is always deleted; on an RSX-11M or M-PLUS system, a file printed under DISPOSE is always saved.

The RMS-11 version of the OTS does not support the DISPOSE= 'PRINT' option; the file is always saved. Likewise, DISPOSE='PRINT' is not supported on RSTS/E; the file is always saved.

2.3.5 INITIALSIZE and EXTENDSIZE

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

INITIALSIZE is effective only at the time a file is created. If you specify EXTENDSIZE when creating a file, the value you specify becomes the default value used by the system to allocate additional storage for the file. If you specify EXTENDSIZE when opening an existing file, the value you specify supersedes any EXTENDSIZE value specified when the file was created, and remains in effect until you close the file.

If the value of INITIALSIZE is positive, the system allocates contiguous space; if the value is negative, the system allocates noncontiguous space.

If there is not enough space available to hold the initial size of a file or to extend a file, an error message is issued.

On RSX–11, if you do not specify an INITIALSIZE value, the system allocates no file storage for data records at the time a file is created; instead, the system allocates file storage dynamically as needed, except on RSTS/E systems. On RSTS/E, if you do nct specify an INITIALSIZE value at file creation, run-time errors may occur. For direct access files, only the file storage actually written is allocated; therefore, a direct-access READ to any point beyond the allocated storage results in an error condition.

2.3.6 KEY

KEY specifies one or more fields to function as keys for accessing records in an indexed file. A KEY specification (not to be confused with "key" specification) has the form:

```
KEY= (kspec [, kspec]...)

KSPEC =.s: e[: dt]
```

s
The starting byte position of the key. (The first byte of a record in FORTRAN–77 is assigned to position 1.

e
The ending byte position of the key.

dt

The key data type: INTEGER, for binary integer keys, or CHARACTER, for character-string keys. (If dt is omitted, the key data type is CHARACTER.)

The data type of a key determines the order in which records are indexed for sorting. The data type of a key is not affected by the formatting you use for a key value at the time you create a record. Usually, however, if you specify integer keys for a formatted file, you should use A-formatting for the key values when creating records in that file. See Section 7.8 for more information on using integer keys.

A key field has a length of e-s+1, where the values of s and e must be such that:

```
1   .LE. (s) .LE. (e) .LE. record-length
1   .LE. (e-s+1) .LE. 255
```

The key length of an integer key must be either 2 or 4, to correspond to INTEGER*2 or INTEGER*4, respectively.

2.3.7 ORGANIZATION

ORGANIZATION specifies the type of organization a file has or is to have. An ORGANIZATION specification has the form:

```
                'SEQUENTIAL'
ORGANIZATION =  'RELATIVE'
                'INDEXED'
```

The default file organization is sequential.

When an existing file is opened, the specified file organization must match the actual file organization. The ORGANIZATION keyword must be specified for relative or indexed files.

2.3.8 READONLY

READONLY specifies that write operations are not to be allowed on the file being opened. The main purpose of READONLY is to allow two or more programs to read a file simultaneously without having to change the protection specified for that file. Changing the protection specified for a file can be hazardous because run-time I/O errors can occur as a result of the default file access privileges being read or written at the same time a file's protection does not permit write access. Therefore, if you want to open a file for the purpose of reading it, but do not want to prevent others from being able to read the same file while you have it open, specify the READONLY keyword. For more information on file sharing, see Section 2.3.11.

2.3.9 RECL (Recordsize)

RECL specifies how much data a record can contain. A RECL specification has the form:

```
RECL
RECORDSIZE = rl
```

The value of rl specifies the length of the logical records in a file. For files that contain fixed-length records, rl specifies the length of each record; for files that contain variable-length records, rl specifies the maximum length of any record; for files containing segmented records, rl specifies the maximum length of any segment.

The value of rl does not include the bytes that the file system requires for maintaining record-length and record-control information; it does, however, include the two segment control bytes, if present.

The value of rl is interpreted as either bytes or storage units (a storage unit consists of four bytes). It is interpreted as bytes if the records are formatted and as storage units if the records are unformatted. Table 2-5 summarizes the maximum values that can be specified for rl for each file organization and record-format combination. Table 2-6 summarizes the default RECL values the system uses when a file is created.

Table 2–5: RECL Value Limits

File Organization	Record Type	
	Formatted (Bytes)	Unformatted (Storage Units)
Sequential	32766	8191
Variable-length records on magnetic tape	9999	2499
Relative	16380	4095
Indexed	16360	4090

Table 2–6: Default RECL Values

Record Type	Size (Bytes)
Formatted	133
Unformatted, fixed-length	128
Unformatted, variable-length	126

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

You must make a RECL specification when you create a file that contains fixed-length records or that has relative organization.

NOTE

> You must specify the Task Builder option MAXBUF if records larger than 133 bytes are to be processed.

2.3.10 RECORDTYPE

RECORDTYPE specifies the structure of (the record type of) the records in a file. A RECORDTYPE specification has the form:

```
                 'FIXED'
RECORDTYPE =     'VARIABLE'
                 'SEGMENTED'
```

RECORDTYPE is particularly useful when you want to override the default record type used in creating a file. The default record type is:

- Fixed if the file organization is indexed or relative, or if the access mode is direct
- Variable if the file organization is sequential and the access mode is formatted sequential
- Segmented if the file organization is sequential and the access mode is unformatted sequential

The default RECORDTYPE value the system uses when accessing an existing file is determined by the record structure of the file, with one exception. In the case of unformatted sequential files containing fixed- or variable-length records you must explicitly override the default (which is 'SEGMENTED') by specifying the appropriate RECORDTYPE value in the OPEN statement. You cannot use an unformatted sequential READ statement to access an unformatted sequential file that contains fixed-length or variable-length records unless you specify the appropriate RECORDTYPE value in an OPEN statement. Files containing segmented records can be accessed only by unformatted sequential I/O statements.

2.3.11 SHARED

SHARED specifies that a file can be accessed by more than one program at a time, or by the same program on more than one logical unit. The forms of mutual accessing, or sharing, permitted depend on the organization of the file and on the I/O system (FCS-11 or RMS–11) in use.

FCS-11 permits multiple readers and a single writer.

RMS–11 permits multiple readers and multiple writers on relative and indexed files. It does not permit multiple writers on sequential files; however, it does permit multiple readers, provided you specify READONLY in all programs that open the files affected.

When you specify the SHARED keyword, other users can access the file with write access. If write-sharing occurs, RMS–11 uses a bucket-locking facility to control operations on the file and ensure that simultaneous write, update, or delete operations on the same record do not occur. See Section 2.6.4 for additional information.

2.3.12 USEROPEN

USEROPEN provides access to features of the supporting I/O system not directly supported by the FORTRAN–77 I/O system. Or, more specifically, USEROPEN allows you to access RMS or FCS capabilities and at the same time retain the ease and convenience of FORTRAN–77 programming.

USEROPEN is intended for experienced users.

For the interface specification for a USEROPEN routine, see the *FORTRAN-77 Object Time System Reference Manual*. Consult the *IAS/RSX-11 I/O Operations Reference Manual* for FCS details. Consult the *RMS–11 MACRO–11 Reference Manual* for RMS details.

2.4 BACKSPACE and ENDFILE Implications

This section describes implications of the BACKSPACE and ENDFILE I/O statements, which are supported only for sequential files.

A BACKSPACE operation cannot be performed on a file that is opened for append access, because under append access the current record count is not available to the FORTRAN–77 I/O system; backspacing from record n is done by rewinding to the start of the file and then performing n-1 successive reads to reach the previous record.

The ENDFILE statement writes an end-file record. Because the concept of an embedded end-file is unique to FORTRAN, the following convention has been adopted: An end-file record is a 1-byte record that contains the octal code 32 (CTRL/Z). An end-file record can be written only to sequentially organized files that are accessed as formatted sequential or unformatted segmented sequential. End-file records should not be written in files that are read by programs written in a language other than FORTRAN.

2.5 FORTRAN-77 Input/Output Using File Control Services (FCS)

File Control Services (FCS-11) is an I/O subsystem that provides sequential and direct access to sequential files. For a detailed discussion of FCS-11, consult the *IAS/RSX-11 I/O Operations Reference Manual.*

2.5.1 OTS/FCS Record Transactions

Records are transferred with FCS record mode macros as follows:

- Sequential input—GET$S
- Sequential output—PUT$S
- Direct input—GET$R
- Direct output—PUT$R

2.5.2 OTS/FCS File Open Conventions

A file or device is opened for I/O activity by the execution of an OPEN statement, or by the execution of an input or output statement if no file/device is already open on the logical unit specified in the statement. The type of FCS open operation invoked is based on the specifications (explicit or implied) in the OPEN statement or on the type of I/O statement, as follows:

Input statement	OPEN$U
Output statement	OPEN$W
OPEN statement	
TYPE='OLD'	OPEN$U
TYPE='NEW'	OPEN$W
TYPE='SCRATCH'	OPEN$W, followed by call to .MRKDL
TYPE='UNKNOWN'	try OPEN$U; if no such file, then OPEN$W.

Files created for formatted input/output (both sequential and direct access) are given the FORTRAN carriage-control attribute (FD.FTN).

2.5.3 FCS Implications of FIND and REWIND

This section describes FCS-specific implications of the FIND and REWIND I/O statements.

A FIND statement is similar to a direct access READ operation with no I/O list and may cause an existing file to be opened; upon execution, it assigns to an associated variable the specified record number.

FCS does not allow error checking with the FIND statement

A REWIND statement is performed as an FCS .POINT operation that specifies positioning at the beginning of the indicated file (block=1, byte=0).

2.5.4 FCS File Sharing

The FCS file system permits files to be simultaneously accessed by two or more tasks.

Two tasks writing to a shared file in which some of the records cross block boundaries may produce undesirable results. That is, because the read, modify, and rewrite sequences performed by two tasks writing to a shared file are asynchronous and independent, a record can occur in which part of the data was written by one task and part by another. In addition, because FCS generally tries to minimize disk activity by postponing a rewrite in case a subsequent read or write can be performed using the in-task buffer image, the disk image of a file may be out-of-date for arbitrary time intervals. This problem of outdatedness may be encountered on both sequential and direct access input/output.

You may encounter a related problem in regard to the logical end-of-file. When a file is extended, the disk description of the logical end-of-file is not updated until the file is closed by all tasks accessing it. Therefore, if one task has opened a file to append new records and has not yet closed it, a second task opening the file to read certain of its records is not able to read any of the new records appended by the first task. Furthermore, it is not able to read any of these new records until the following has occurred: The first task has closed the file; the second task has closed the file; the second task has reopened the file.

When using shared files, you must pay careful attention to the intertask coordination required for reliable performance.

2.6 FORTRAN–77 Input/Output Using Record Management Services (RMS)

Record Management Services (RMS–11) is an I/O system that supports sequential and direct access to sequential and relative files. For a detailed discussion of RMS–11, consult the *RMS–11 MACRO–11 Reference Manual* and the *RMS–11 User's Guide*. The *RMS–11 User's Guide* contains useful information on RMS–11 file structures and ways to improve performance. Note, however, that the RMS–11 features that are a part of FORTRAN–77 are a subset of the total facilities discussed in the *RMS–11 User's Guide*; all RMS features, however, are available through USEROPEN.

2.6.1 OTS/RMS Record Transactions

To read records under RMS, READ statements use the RMS $GET macro; to write to records, WRITE statements use the RMS $PUT macro to add new records and the RMS $UPDATE macro to rewrite existing records in a direct access sequential file.

To update a record in an indexed file, the REWRITE statement uses the RMS $UPDATE macro.

To delete records, the DELETE statement uses the RMS $DELETE macro. You cannot DELETE records in a sequential file.

2.6.2 OTS/RMS File Open Conventions

A file or device is opened for I/O activity by the execution of an OPEN statement or by the execution of an input or output statement. The type of open operation invoked is based on the specifications in the OPEN statement or on the type of I/O statement, as follows:

Input statement	$OPEN
Output statement	$CREATE
OPEN statement	
TYPE='OLD'	$OPEN
TYPE='NEW'	$CREATE
TYPE='SCRATCH'	$CREATE with FB$TMD set
TYPE='UNKNOWN'	OPEN$; if no such file, then $CREATE.

2.6.3 RMS Implications of FIND, REWIND, UNLOCK

This section describes RMS-specific implications of the FIND, REWIND, and UNLOCK statements.

A FIND statement is similar to a direct access READ statement with no I/O list and may cause an existing file to be opened. The RMS $FIND macro is executed to locate and lock the specified record.

A REWIND statement results in a call to the RMS $REWIND macro.

The UNLOCK statement unlocks the bucket currently locked on the specified logical unit by executing the RMS $FREE macro. If no record is locked, the operation has no effect. See Section 2.6.4 for details on file sharing and using the UNLOCK statement.

2.6.4 RMS File Sharing

You can write-share relative and indexed files, but not sequential files.

If a program has write access to a shared file, RMS–11 locks every bucket accessed by a successful READ or FIND statement until another I/O operation is performed on the same logical unit. If a program attempts to access a record that RMS has locked, FORTRAN run-time error "SPECIFIED RECORD LOCKED" is reported.

To ensure the greatest flexibility at run time, you should always anticipate the possibility that any record you attempt to access might be locked by another logical unit in your own program or by a logical unit in another program. You can be properly prepared by employing the following procedures when you write programs.

If you are using a single logical unit to access a file and you encounter the record-locked error, you can reexecute the I/O statement that failed until RMS–11 indicates successful completion.

If you are using multiple logical units to access a file, you cannot simply reexecute the I/O statement that failed. One of your other logical units may have locked the target bucket; therefore, you could place your program in an infinite loop if you were to continue to execute the same statement. Instead, you should first execute an UNLOCK statement for all other logical units having access to the same file in your program. You can then safely reexecute the I/O statement until RMS–11 indicates successful completion.

Never retain a lock on a bucket longer than necessary. For example, when you execute a successful READ or FIND statement, you cause RMS–11 to lock a bucket; therefore, you should immediately execute an UNLOCK on the logical unit so that RMS–11 will unlock the bucket.

The following program segment demonstrates the programming techniques you should use for shared files. The program attempts to access a record whose key value is contained in the byte array KEYVAL.

```
10      READ (IDXLUN, KEY=KEYVAL, ERR=20) DATA
        UNLOCK (IDXLUN)
        .
        .
        (process record)
        .
        .
        .
20      CALL ERRSNS (IERR)
        IF (IERR .EQ. 52) GO TO 10
        TYPE *, 'ERROR READING INDEXED FILE', IERR
        STOP
        END
```

2.6.5 Task Building with RMS

RMS–11 is a set of file access routines that execute as part of a task. The Task Builder resolves references to these routines in either an object library (LB:[1,1]RMSLIB.OLB or LB:RMSLIB.OLB on RSTS/E) or a resident library (RMSRES or RMSSEQ).

Because these routines add from 8K bytes to 44K bytes to the size of a task, you may need to overlay the RMS portion of a task. A series of standard ODL files that describe disk-resident overlays requiring different amounts of space is provided. Table 2–7 shows the size of the RMS portion and the RMS features included for each standard ODL file.

Table 2–7: RMS File System Libraries

File Name[1]	Approximate Addition	Features Included
LB:[1,1]RMS11S.ODL	8K bytes	Sequential and relative organizations
LB:[1,1]RMS11X.ODL	9K bytes	Sequential, relative and indexed organizations
LB:[1,1]RMS12X.ODL	12K bytes	Sequential, relative and indexed organizations (in fewer overlay segments than RMS11X)

[1] Do not include [1,1] on RSTS/E systems

A prototype ODL file, LB:[1,1]RMS11.ODL, is also provided (LB:RMS11.ODL on RSTS/E). This file is similar to RMS11X.ODL, but it contains comments and instructions to aid you in optimizing the overlay structure to accommodate your particular task requirements. Refer to the *RMS–11 User's Guide* for information on optimizing the overlay structure.

The standard RMS ODL files are incorporated into a program ODL file as follows:

The factor RMSROT (which is defined in the RMS ODL file, that is, in RMS11S.ODL, RMS11X.ODL, and RMS12X.ODL) must be added to the task root segment. The factor RMSALL (which is also defined in the RMS ODL files) should be added as an RMS co-tree root segment. For example:

```
.ROOT    MAIN-RMSROT,RMSALL        ;RMS co-tree
```

The following ODL file builds the same overlaid program as described in Section 1.4; it incorporates an overlaid RSX–11M OTS and the 12K-byte version of RMS as a co-tree. On RSTS/E, [1,1] would not be included. (See Section 5.4.8 for more information on overlaying the FORTRAN–77 OTS.)

```
          .ROOT  MAIN-OTSROT-RMSROT-OVL,OTSALL,RMSALL
OVL:      .FCTR  *(PRE,PROC,POST)
@LB:[1,1]RMS11M
@LB:[1,1]RMS12X
          .END
```

NOTE

> The FORTRAN–77 OTS and RMS must both be set up as co-trees (as shown above) or not overlaid at all.

You may also be able to use an RMS–11 shared resident library (RMSRES) if your system supports one. Using RMSRES requires 16K bytes of address space, but significantly reduces both task-build time and execution time.

You can include the RMS shared library RMSRES in your task by using the following procedure:

- Specify LB:[1,1]RMSRLX.ODL (LB:RMSRLX.ODL on RSTS/E) as the indirect RMS ODL file within your ODL file.
- Include LIBR= RMSRES:RO as a task-build option.

You can include the RMS shared library for sequential organization, RMSSEQ, in your task by using the following procedure:

- Specify LB:[1,1]RMSRLS.ODL (LB:RMSRLS.ODL on RSTS/E) as the indirect RMS ODL file within your ODL file.
- Include LIBR= RMSSEQ:RO as a task-build option.

Chapter 3

PDP–11 FORTRAN–77 Operating Environment

This chapter discusses aspects of the PDP–11 FORTRAN–77 compiler and OTS operating environment. Information is provided on the following:

- The PDP–11 calling sequence convention
- FORTRAN program sections
- FORTRAN COMMON blocks and RSX–11 system common blocks
- FORTRAN–77 OTS shared libraries
- FORTRAN–77 OTS error processing
- Compiler listing-file format

3.1 FORTRAN–77 Object Time System

The FORTRAN–77 Object Time System (OTS) is composed of the following routines:

- Math routines, including the FORTRAN–77 library functions and other arithmetic routines (for example, exponentiation routines)
- Miscellaneous utility routines (ASSIGN, DATE, ERRSET, and so forth)
- Routines that handle FORTRAN–77 input/output
- Error-handling routines that process arithmetic errors, I/O errors, and system errors
- Miscellaneous routines required by the compiled code

The FORTRAN–77 OTS is a collection of many small modules that allows you to omit unnecessary routines during task-building. For example, if a program performs only sequential formatted I/O, none of the direct-access I/O routines are included in the task.

3.2 FORTRAN–77 Calling Sequence Convention

The PDP–11 FORTRAN–77 calling sequence convention is compatible with all PDP–11 processor options and provides both reentrant and nonreentrant forms.

3.2.1 The Call Site

The MACRO–11 form of the call is:

```
 ; IN INSTRUCTION-SPACE

          MOV #LIST,R5    ;ADDRESS OF ARGUMENT LIST TO
                          ;REGISTER 5
          JSR PC,SUB      ;CALL SUBROUTINE
             .
             .
             .

; IN DATA-SPACE

LIST:     .BYTE N,0       ;NUMBER OF ARGUMENTS
          .WORD ADR1      ;FIRST ARGUMENT ADDRESS
             .
             .
             .
          .WORD ADRN      ;N'TH ARGUMENT ADDRESS
```

The argument list must reside in DATA-SPACE and, except for subprograms and statement label arguments, all addresses in the list must also refer to DATA-SPACE. The argument list itself cannot be modified by the called program.

The byte at address LIST+1 should be considered undefined and not referenced. This byte is reserved for use as defined by DIGITAL.

The called program is free to refer to the arguments indirectly through the argument list. This argument-passing mechanism is known as call-by-reference.

3.2.2 Return

Control is returned to the calling program by restoring (if necessary) the stack pointer to its value on entry and executing the following:

```
RTS PC
```

3.2.3 Return Value Transmission

Function subprograms return a single result in the processor general registers. The register assignments for returning the different variable types are:

Type	Result
INTEGER*2	
LOGICAL*1	R0
LOGICAL*2	
INTEGER*4	R0—low-order result
LOGICAL*4	R1—high-order result
Real	R0—high-order result
	R1—low-order result
	R0—highest-order result
Double	R1 –
Precision	R2 –
	R3—lowest-order result
Complex	R0—high-order real result
	R1—low-order real result
	R2—high-order imaginary result
	R3—low-order imaginary result

3.2.4 Register Usage Conventions

Before making a call, a calling program must save any values in general-purpose registers R0 through R4 that it needs after a return from a subprogram. After a return, a calling program cannot assume that the argument list pointer value in register R5 is valid.

Conventions for floating point registers are similar to those for general-purpose registers. If a Floating Point Processor (FP11) or the floating point microcode option (KEF11A) is present on a system, the calling program must save and restore any floating point registers in use by a calling program. The calling program cannot assume that the floating point status bits I/L (integer/long integer) or F/D (floating/double precision) are restored by the called routine.

A subprogram that is called by a FORTRAN-77 program can freely use processor registers R0-R5, FPP registers F0-F5, and the FPP status register. When returning from a subroutine (when the RTS PC is executed), the initial (routine entry) value must be restored to the contents of the processor hardware stack pointer SP.

3.2.5 Nonreentrant Example

In nonreentrant forms, the argument list can be placed either in line with the call or out of line in an impure data section. (The latter is recommended and illustrated here, and is the form produced by the FORTRAN-77 compiler.) Example 3-1 illustrates assembly language code implementing a small FORTRAN-77 FUNCTION subprogram that uses the nonreentrant form of a call. Note that the nonreentrant form is shorter and generally faster than the reentrant form because addresses of simple variables can be assembled into the argument list.

Example 3–1: Call Sequence Conventions: Nonreentrant Example

```
          INTEGER FUNCTION FNC(I,J)
          INTEGER FNC1
          FNC=FNC1(I**,5)+I
          RETURN
          END

          .PSECT
          .GLOBL     FNC,FNC1
FNC:      MOV        R5,-(SP)       ;SAVE ARG LIST POINTER
          MOV        @2(R5),-(SP)   ;FORM I+J ON STACK
          ADD        @4(R5),@SP
          MOV        SP,LIST+2      ;ADDRESS OF I+J TO
                                    ;ARG LIST

          MOV        #LIST,R5
          JSR        PC,FNC1
          ADD        #2,SP          ;DELETE TEMPORARY I+J
          MOV        (SP)+,R5       ;RESTORE R5
          ADD        @2(R5),R0      ;ADD I TO FNC1 RESULT
          RTS        PC             ;RETURN VALUE IN R0

          .PSECT     DATA           ;DATA AREA
LIST:     .BYTE      2,0            ;TWO ARGUMENTS
          .WORD      0              ;DYNAMICALLY FILLED IN
          .WORD      LIT5           ;ADDRESS OF CONSTANT 5
;
LIT5:     .WORD      5,0            ;CONSTANT 5
          .END
```

3.2.6 Reentrant Example

The PDP–11 FORTRAN–77 calling convention has a reentrant form in which the argument list is constructed at run time on the execution stack. Note that the argument addresses must be pushed backwards on the stack to be correctly arranged in memory for the subroutine that uses the list. Basically, the technique consists of:

```
MOV          #ADRn,-(SP)      ;ADDRESS OF NTH ARGUMENT
               .
               .
               .
```

```
MOV          #ADR2,-(SP)
MOV          #ADR1,-(SP)      ;ADDRESS OF 1ST ARGUMENT
MOV          #n,-(SP)         ;NUMBER OF ARGUMENTS
MOV          SP,R5
JSR          PC,SUB           ;CALL SUBROUTINE
ADD          #2*n+2,SP        ;DELETE ARGUMENT LIST
```

Example 3–2 illustrates assembly language code that uses reentrant call forms for the same example shown in Example 3–1.

The FORTRAN-77 compiler does not produce reentrant call forms.

Example 3–2: Call Sequence Convention: Reentrant Example

```
          INTEGER FUNCTION FNC(I,J)
          INTEGER FNC1
          FNC=FNC1(I+J,5)+I
          RETURN
          END

          .PSECT
          .GLOBL      FNC,FNC1
FNC:      MOV         R5,-(SP)       ;SAVE ARG LIST POINTER
          MOV         @2(R5),-(SP)   ;FORM I+J
          ADD         @4(R5),@SP     ;ON STACK
          MOV         SP,R4          ;REMEMBER WHERE
          MOV         #CON5,-(SP)    ;BUILD ARG LIST ON STACK
          MOV         R4,-(SP)       ;ADDRESS OF TEMPORARY
          MOV         #2,-(SP)       ;ARGUMENT COUNT
          MOV         SP,R5          ;ADDRESS OF LIST TO R5
          JSR         PC,FNC1        ;CALL FNC1
          ADD         #10,SP         ;DELETE ARG LIST AND TEMP I+J
          MOV         (SP)+,R5       ;RESTORE ARG LIST POINTER
          ADD         @2(R5),R0      ;ADD I TO RESULT OF FNC1
          RTS         PC             ;RETURN RESULT IN R0

          .PSECT      DATA           ;DATA AREA
CON5:     .WORD       5,0
          .END
```

3.2.7 Null Arguments

Null arguments are represented in an argument list with an address of -1 (177777 octal). This address is chosen to ensure that using null arguments in calling routines not prepared to handle null arguments will result in an error when the routine is called at execution time. The errors most likely to occur are illegal memory references and/or word reference to odd byte addresses.

Note that null arguments are included in the argument count, as follows:

FORTRAN Statement		
CALL SUB	.BYTE	0,0
CALL SUB( )	.BYTE .WORD	1,0 -1
CALL SUB(A,)	.BYTE .WORD .WORD	2,0 A -1
CALL SUB(,B)	.BYTE .WORD .WORD	2,0 -1 B

3.3 Program Sections

Program sections (PSECTs) are named segments of code and/or data. Attributes associated with each program section (see Table 3–1) direct the Task Builder when the Task Builder is combining separately compiled FORTRAN program units, assembly language modules, and library routines into an executable task image.

3.3.1 Compiled-Code PSECT Usage

The compiler uses PSECTs to organize compiled output into the following six sections:

1. Section $CODE1 contains all of the executable code for a program unit.
2. Section $PDATA contains pure data, such as constants, that cannot change during program execution.
3. Section $IDATA contains impure data, such as argument lists, that can change during program execution.
4. Section $VARS contains storage allocated for variables and arrays used in a program.
5. Section $TEMPS contains temporary storage allocated by the compiler.
6. Section $SAVE contains global storage for entities specified in a SAVE statement.

The attributes associated with each of these sections are shown in Table 3–1.

Table 3–1: Program Section Attributes

Section Name	Attributes
$CODE1	RW, I, LCL, REL, CON
$PDATA	RW, D, LCL, REL, CON
$IDATA	RW, D, LCL, REL, CON
$VARS	RW, D, LCL, REL, CON
$TEMPS	RW, D, LCL, REL, CON
$SAVE	RW, D, GBL, REL, CON, SAV

NOTE

The RO/RW attributes for the sections $CODE1 and $PDATA are controlled by the compiler /RO command qualifier.

Section attributes are as follows:

RW, RO	Read/write, read only
I, D	Instructions, data
CON, OVR	Concatenated, overlaid
LCL, GBL	Local within overlay segment, global across segments
SAV	Unconditionally place PSECT in root segment

Because FORTRAN–77 programs contain statically allocated impure storage, compiled object modules are not reentrant and cannot be included in a shareable library.

Virtual arrays are allocated into a special control section, $VIRT, that the Task Builder allocates into the mapped array area of a task.

3.3.2 FORTRAN COMMON and RSX–11 System Common

You can indicate that a common block in a task is to reference a system global common block of the same name. You can do this, at task-build time, with the Task Builder option:

```
COMMON = name:access[:apr]
```

where name is any valid common block name, access may be either RO *for read-only access or RW for read/write access, and the optional element* apr is an integer from 1 to 7 that specifies the first Active Page Register. If the common block defined in the user task is larger than the corresponding system global common block, a fatal task-build error results.

If a task attempts to initialize any storage in a common block by using DATA statements, a fatal task-build error results.

Storage for a common block is placed into a PSECT of the same name as that of the common block. PSECTs used for common blocks are given the attributes RW, D, GBL, REL, OVR, and, for saved named common blocks and blank common, SAV. (The /F77 switch must be set for the blank common block PSECT to have the SAV attribute; named common block PSECTS have the SAV attribute under either /F77 or /NOF77.) For example, the statement

```
COMMON /X/A,B,C
```

produces the equivalent of the following MACRO-11 code:

```
   .PSECT   X,RW,D,GBL,REL,OVR, SAV
A: .BLKW 2
B: .BLKW 2
C: .BLKW 2
```

A blank common uses the section name .$$$$. Therefore, under /F77 the statement

```
COMMON T,U,V
```

produces the equivalent of:

```
    .PSECT   .$$$$.,RW,D,GBL,REL,OVR,SAV
T:  .BLKW 2
U:  .BLKW 2
V:  .BLKW 2
```

When named PSECTs with the OVR attribute are combined by the Task Builder, all PSECTs with the same name are allocated to begin at the same address. The resulting PSECT has the length of the largest of the combined PSECTs.

An example of common communication between a FORTRAN-77 main program and an assembly language subroutine is shown in Examples 3-3 and 3-4. In the example, the variable ISTRNG in blank common is filled with Hollerith data. This variable is copied to OSTRNG (with space characters removed) in the labeled common DATA, and the actual length is returned in the variable LEN.

Note that one word is allocated for each integer in the assembly language subroutine; this allocation convention is necessary for compatibility with FORTRAN storage allocation under the default /NOI4 setting for compilation.

Example 3-3 shows the FORTRAN main program compiled under the /NOI4 option. The assembly language subroutine COMPRS is shown in Example 3-4.

Example 3–3: Establishing a FORTRAN COMMON Area and Assembly Language Subroutine CALL

```
        LOGICAL*1 ISTRNG(80),OSTRNG(80)
        COMMON ISTRNG
        COMMON /DATA/ LEN, OSTRNG
C       GET INPUT STRING
C
        READ 1, ISTRNG
1       FORMAT( 80A1 )
C       COMPRESS THE STRING
C
        CALL COMPRS
C       TYPE OUT THE COMPRESSED STRING
C
        TYPE 2, LEN, (OSTRNG(I),I=1,LEN)
2       FORMAT(1X,I3,6X,80A1)
        END
```

3.3.3 OTS PSECT Usage

All OTS modules consist of at least two program sections: $$OTSI and $$OTSD. Section $$OTSI contains pure-code sequences and section $$OTSD contains pure-data information.

The OTS module $OTV declares the following sections that are used as impure working storage by the OTS:

- Section $$AOTS contains a general work area.
- Section $$DEVT contains storage for each FORTRAN logical unit. The size of $$DEVT is determined by the Task Builder UNITS option.
- Section $$FSR1 contains storage for I/O buffers and file-system control blocks. The size of $$FSR1 is determined by the Task Builder option ACTFIL.
- Section $$IOB1 contains storage for the FORTRAN–77 input/output record buffer. The size of $$IOB1 is determined by the Task Builder option MAXBUF.
- Section $$OBF1 contains storage for holding the compiled form of object-time formats. The size of $$OBF1 is determined by the Task Builder FMTBUF option.

Example 3-4: Use of FORTRAN COMMON Area by Assembly Language Subroutine

```
        .TITLE   COMPRS
        .IDENT   /01/

       ;COMPRESS THE HOLLERITH STRING IN BLANK COMMON
       ;COPYING THE STRING TO LABELLED COMMON DATA AND
       ;RETURNING THE ACTUAL LENGTH AS WELL.

        .PSECT   .$$$$.,D,GBL,OVR
I:      .BLKB    80.                ; INPUT BUFFER

        .PSECT   DATA,D,GBL,OVR
L:      .BLKW    1                  ; ACTUAL LENGTH
O:      .BLKB    80.                ; OUTPUT BUFFER

        .PSECT
COMPRS::
        MOV      #I,RO              ; INPUT POINTER
        MOV      #O,R1              ; OUTPUT POINTER
        MOV      #80.,R2            ; INPUT LENGTH
        CLR      L                  ; OUTPUT LENGTH
1$:     MOVB     (RO)+,R3           ; GET INPUT CHARACTER
        CMPB     #' ,R3             ; IS THIS CHAR A SPACE?
        BEQ      2$ <               ; IGNORE IF SO
        MOVB     R3,(R1)+           ; OUTPUT THE CHARACTER
        INC      L                  ; COUNT THE CHARACTER
2$:     DEC      R2                 ; COUNT DOWN THE INPUT
        BGT      1$                 ; LOOP IF MORE DATA
        RTS      PC
        .END
```

The handling and conversion routines for formatted records are contained in the following sections: $$FIOC, $$FIOD, $$FIO2, $$FIOI, $$FIOL, $$FIOZ, $$FIOS, and $$FIOR. Special conventions are used so that the conversion routines are loaded only if they are required by FORMAT statements in a source program.

3.4 OTS and Resident (Shareable) Libraries

Each module of the FORTRAN-77 OTS (with the exception of modules $OTV, LICSB$, $ORGSQ, $ORGRL, and $ORGIX) consists only of code and data that is pure and shareable. Consequently, all or any part of the OTS can be built into a resident (shareable) library or included in another resident library. However, the OTS does not consist of position-independent code (PIC) and cannot, therefore, be included in a resident library that does consist of PIC. In particular, the OTS cannot be included in resident libraries SYSRES, FCSRES, or RMSRES of the I/O system, because each of these libraries consists of PIC.

Procedures for building a FORTRAN-77 OTS resident library are described in Chapter 13 of the *PDP-11 FORTRAN-77 Object Time System Reference Manual.*

3.5 OTS Error Processing

The Object Time System detects certain errors in a program (for example, I/O, arithmetic, and invalid argument errors) and reports these errors on the user's terminal. An error-control table within the OTS then determines what action the system is to take for each error reported; for example, it may call for the system to terminate the task. The default action for each FORTRAN-specific error is shown in Table 3-2 (in Section 3.5.1.3).

Three system subroutines (ERRSNS, ERRTST, and ERRSET) are provided to enable you to control OTS error processing: that is, to obtain information on specific errors and/or to specify an action to be taken when a specific error occurs.

The ERRSNS subroutine provides you with information about the error that has most recently occurred during program execution. It also provides detailed information on errors detected by the file system (FCS-11 or RMS-11).

The ERRTST subroutine allows you to test for the occurrence of a specific error during program execution.

The ERRSET subroutine allows you to modify the continuation action the system is to take when an error is detected by the OTS. In many cases, the particular continuation action to be taken may be changed from the one specified in the error-control table (see Table 3-2).

The subroutines ERRSNS, ERRTST, and ERRSET are described in detail in Appendix D. OTS error codes and the format of the OTS diagnostic messages are shown in Appendix C.

3.5.1 Recovering from OTS-Detected Errors

You can use three methods to control recovery from errors detected by the OTS:

- ERR= and END= transfers
- The ERRSNS subroutine
- The ERRSET subroutine

The following three sections discuss these methods.

3.5.1.1 Using ERR= and END= Transfers

By including an ERR=label or END=label specification in an I/O statement, you can transfer control to error-processing code or to any other desired point in a program. If you use an END= or ERR= specification to process an I/O error, execution continues at the statement specified by a label. However, if you do not use an END= or ERR= specification to process an I/O error, the system by default prints an error message and halts execution.

For example, suppose the following statement is in your program:

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

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

When an ERR= transfer occurs, file status and record position become undefined.

You can use the END=label specification to handle an end-of-file condition. For example, if an end-of-file condition is detected while the statement

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

is being executed, control transfers to statement 550.

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

3.5.1.2 Using the ERRSNS Subroutine

You can use the ERRSNS system subroutine to process errors as they are encountered by a program. When one of the errors listed in Table 3-2 occurs in a program, you can obtain the number of the error by calling the ERRSNS subroutine; then, in most situations, you can provide code to react to this number.

To determine the number of an error, use the ERRSNS routine as demonstrated in the following example:

```
     CHARACTER*40 FILN
10   ACCEPT 1, FILN

1    FORMAT (A)
     OPEN (UNIT=INF, STATUS='OLD', FILE=FILN, ERR=100)
          .
          . (process input file)
          .

100  CALL ERRSNS(IERR)
     IF (IERR .EQ. 43) THEN
        TYPE *, 'FILE NAME WAS INCORRECT; ENTER NEW FILE NAME'
     ELSE IF(IERR .EQ. 29) THEN
        TYPE *, 'FILE DOES NOT EXIST; ENTER NEW FILE NAME'
     ELSE
        TYPE *, 'FAILURE ON INPUT FILE; ERROR=', IERR
     ENDIF
     STOP
     END
```

In this example, the OPEN statement contains an ERR=100 specification that causes a branch to the ERRSNS subroutine if an error occurs during execution of the OPEN. The ERRSNS subroutine returns an error-number value in the integer variable IERR. The program then uses the value of IERR to print a message that explains the nature of the error and to determine whether the program should continue.

3.5.1.3 Using the ERRSET Subroutine

You can alter the default continuation action to be taken upon OTS detection of a particular error by using the ERRSET subroutine.

Processing each of the errors detected by the OTS is controlled by six control bits associated with each error. These bits are preset (see Table 3-2); however, you may alter some of the initial settings—and thereby the continuation action to be taken upon the detection of a particular error—by using the ERRSET subroutine.

The six control bits and what they control are as follows:

1. Continuation Bit—If the Continuation Bit is not set, the task encountering the error exits. If this bit is set, the task continues (if the next two conditions permit continuation).
2. Count Bit—If the Count Bit is set, the error encountered is counted against the task error-count limit unless an ERR=transfer is specified. If the error-count limit is exceeded, the task exits.
3. Continuation Type Bit—The Continuation Type Bit provides for one of the following two types of action for a particular error:
   a. Return to the routine that reported the error, for appropriate recovery action, then proceed.
   b. Take an ERR= transfer in an I/O statement. (If the Continuation Type Bit specifies an ERR= transfer, and no ERR=label was included in the I/O statement, the task exits).

Each of the error-control-bit checks above must be satisfied for the task to continue.

4. Log Bit—If a task continues after an error is encountered (that is, if continuation is permitted by each of the above three control bits), then the Log Bit is tested. If the Log Bit is set, an error message is produced before the task continues; if the Log Bit is not set, the task continues without a message.

If processing any of the first three control bits does not permit continuation, the task exits and the system prints an error message.

Two additional control bits are used to specify the acceptability of arguments to the ERRSET subroutine.

5. Return Permitted Bit—If the Return Permitted Bit is set, ERRSET may set the Continuation Type Bit to specify a return.

6. ERR= Permitted Bit—If the ERR= Permitted Bit is set, ERRSET may set the Continuation Type Bit to specify that an ERR= transfer is to occur.

At least one of these two additional bits must be set in order for the Continuation Bit to be set.

All four of the possible combinations of these two bits occur in the OTS; however, most errors occur as the following:

- I/O errors that generally permit ERR= continuation type but not return continuation
- Errors that permit return continuation but not ERR= transfer continuation (even if they occur during I/O statement processing)

Notable exceptions are the synchronous system-trap errors (3 through 10) and the recursive I/O error (40), all of which always result in task termination. The format processing and format conversion errors (59, 61, 63, 64, 68) allow both types of continuation.

The initial setting of all six control bits—the two permitted bits as well as the Continuation Bit, the Count Bit, the Continuation Type Bit, and the Log Bit—is shown in Table 3-2. You can use the ERRSET subroutine to change the settings for CONTINUE?, COUNT?, CONTINUE TYPE, and LOG?. The ERRSET subroutine is described in detail in Appendix D.

Table 3–2: Initial Error Control Bit Settings

Error			Continue		Permitted		
Number	**Continue?**	**Count?**	**Type**	**Log?**	**Err =?**	**Return?**	
1	NO	NO	FATAL	YES	NO	NO	INVALID ERROR CALL
2	NO	NO	FATAL	YES	NO	NO	TASK INITIALIZATION FAILURE
3	NO	NO	FATAL	YES	NO	NO	ODD ADDRESS TRAP (SST0)
4	NO	NO	FATAL	YES	NO	NO	SEGMENT FAULT (SST1)
5	NO	NO	FATAL	YES	NO	NO	T-BIT OR BPT TRAP (ST2)
6	NO	NO	FATAL	YES	NO	NO	IOT TRAP (SST3)
7	NO	NO	FATAL	YES	NO	NO	RESERVED INSTRUCTION TRAP . . .

Table 3–2 (Cont.): Initial Error Control Bit Settings

Error			Continue		Permitted		
Number	Continue?	Count?	Type	Log?	Err =?	Return?	
8	NO	NO	FATAL	YES	NO	NO	NON-RSX EMT TRAP (SST5)
9	NO	NO	FATAL	YES	NO	NO	TRAP INSTRUCTION TRAP (SST6)
10	NO	NO	FATAL	YES	NO	NO	PDP-11/40 FIS TRAP (SST7)
11	NO	NO	FATAL	YES	NO	NO	FPP HARDWARE FAULT
12	NO	NO	FATAL	YES	NO	NO	FPP ILLEGAL OPCODE TRAP
13	NO	NO	FATAL	YES	NO	NO	FPP UNDEFINED VARIABLE TRAP
14	NO	NO	FATAL	YES	NO	NO	FPP MAINTENANCE TRAP
20	YES	YES	ERR=	YES	YES	NO	REWIND ERROR
21	YES	YES	ERR=	YES	YES	NO	DUPLICATE FILE SPECIFICATIONS
22	YES	YES	ERR=	YES	YES	NO	INPUT RECORD TOO LONG
23	YES	YES	ERR=	YES	YES	NO	BACKSPACE ERROR
24	YES	YES	ERR=	YES	YES	NO	END-OF-FILE DURING READ
25	YES	YES	ERR=	YES	YES	NO	RECORD NUMBER OUTSIDE RANGE
26	YES	YES	ERR=	YES	YES	NO	MODE NOT SPECIFIED
27	YES	YES	ERR=	YES	YES	NO	MORE THAN ONE RECORD IN I/O . . .
28	YES	YES	ERR=	YES	YES	NO	CLOSE ERROR
29	YES	YES	ERR=	YES	YES	NO	NO SUCH FILE
30	YES	YES	ERR=	YES	YES	NO	OPEN FAILURE
31	YES	YES	ERR=	YES	YES	NO	MIXED FILE ACCESS MODES
32	YES	YES	ERR=	YES	YES	NO	INVALID LOGICAL NUMBER

Table 3–2 (Cont.): Initial Error Control Bit Settings

Error			Continue		Permitted		
Number	Continue?	Count?	Type	Log?	Err =?	Return?	
33	YES	YES	ERR=	YES	YES	YES	ENDFILE ERROR
34	YES	ERR=	YES	YES	NO	UNIT ALREADY OPEN	
35	YES	YES	ERR=	YES	YES	NO	SEGMENTED RECORD FORMAT ERROR
36	YES	YES	ERR=	YES	YES	NO	ATTEMPT TO ACCESS NON-EXISTENT . . .
37	YES	YES	ERR=	YES	YES	YES	INCONSISTENT RECORD . . .
38	YES	YES	ERR=	YES	YES	NO	ERROR DURING WRITE
39	YES	YES	ERR=	YES	YES	NO	ERROR DURING READ
40	NO	NO	FATAL	YES	NO	NO	RECURSIVE I/O OPERATION
41	YES	YES	ERR=	YES	YES	NO	NO BUFFER ROOM
42	YES	YES	ERR=	YES	YES	NO	NO SUCH DEVICE
43	YES	YES	RETURN	YES	NO	YES	FILE NAME SPECIFICATION ERROR
44	YES	YES	ERR=	YES	YES	NO	INCONSISTENT RECORD TYPE
45	YES	YES	ERR=	YES	YES	NO	KEYWORD VALUE ERROR IN OPEN . . .
46	YES	YES	ERR=	YES	YES	NO	INCONSISTENT OPEN /CLOSE . . .
47	YES	YES	ERR=	YES	YES	NO	WRITE TO READONLY FILE
48	YES	YES	ERR=	YES	YES	NO	UNSUPPORTED I/O OPERATION
49	YES	YES	ERR=	YES	YES	NO	INVALID KEY SPECIFICATION
50	YES	YES	ERR=	YES	YES	NO	INCONSISTENT KEY CHANGE OR . . .

Table 3–2 (Cont.): Initial Error Control Bit Settings

Error Number	Continue?	Count?	Continue Type	Log?	Permitted Err =?	Return?	
51	YES	YES	ERR=	YES	YES	NO	INCONSISTENT FILE ORGANIZATION
52	YES	YES	ERR=	NO	YES	NO	SPECIFIED RECORD LOCKED
53	YES	YES	ERR=	YES	YES	NO	NO CURRENT RECORD
54	YES	YES	ERR=	YES	YES	NO	REWRITE ERROR
55	YES	YES	ERR=	YES	YES	NO	DELETE ERROR
56	YES	YES	ERR=	YES	YES	NO	UNLOCK ERROR
57	YES	YES	ERR=	YES	YES	NO	FIND ERROR
59	YES	NO	ERR=	YES	YES	YES	LIST-DIRECTED I/O SYNTAX ERROR
60	YES	YES	ERR=	YES	YES	NO	INFINITE FORMAT LOOP
61	YES	YES	ERR=	YES	YES	YES	FORMAT/VARIABLE-TYPE MISMATCH
62	YES	YES	ERR=	YES	YES	NO	SYNTAX ERROR IN FORMAT
63	YES	NO	RETURN	YES	YES	YES	OUTPUT CONVERSION ERROR
64	YES	YES	ERR=	YES	YES	YES	INPUT CONVERSION ERROR
65	YES	YES	ERR=	YES	YES	NO	FORMAT TOO BIG FOR 'FMTBUF'
66	YES	YES	ERR=	YES	YES	NO	OUTPUT STATEMENT OVERFLOWS . . .
67	YES	YES	ERR=	YES	YES	NO	RECORD TOO SMALL FOR I/O LIST
68	YES	YES	ERR=	YES	YES	YES	VARIABLE FORMAT EXPRESSION . . .
70	YES	YES	RETURN	YES	NO	YES	INTEGER OVERFLOW
71	YES	YES	RETURN	YES	NO	YES	INTEGER ZERO DIVIDE
72	YES	YES	RETURN	YES	NO	YES	FLOATING OVERFLOW

Table 3–2 (Cont.): Initial Error Control Bit Settings

Error			Continue		Permitted		
Number	**Continue?**	**Count?**	**Type**	**Log?**	**Err =?**	**Return?**	
73	YES	YES	RETURN	YES	NO	YES	FLOATING ZERO DIVIDE
74	YES	NO	RETURN	NO	NO	YES	FLOATING UNDERFLOW
75	YES	YES	RETURN	YES	NO	YES	FPP FLOATING TO INTEGER . . .
80	YES	YES	RETURN	YES	NO	YES	WRONG NUMBER OF ARGUMENTS
81	YES	YES	RETURN	YES	NO	YES	INVALID ARGUMENT
82	YES	YES	RETURN	YES	NO	YES	UNDEFINED EXPONENTIATION
83	YES	YES	RETURN	YES	NO	YES	LOGARITHM OF ZERO OR NEGATIVE . . .
84	YES	YES	RETURN	YES	NO	YES	SQUARE ROOT OF NEGATIVE VALUE
86	YES	YES	RETURN	YES	NO	YES	INVALID ERROR NUMBER
91	YES	NO	RETURN	NO	NO	YES	COMPUTED GOTO OUT OF RANGE
92	YES	YES	RETURN	YES	NO	YES	ASSIGNED LABEL NOT IN LIST
93	YES	YES	RETURN	YES	NO	YES	ADJUSTABLE ARRAY DIMENSION . . .
94	YES	YES	RETURN	YES	NO	YES	ARRAY REFERENCE OUTSIDE ARRAY
95	NO	NO	FATAL	YES	NO	NO	INCOMPATIBLE FORTRAN OBJECT . . .
96	NO	NO	FATAL	YES	NO	NO	MISSING FORMAT CONVERSION . . .
97	NO	NO	FATAL	YES	NO	NO	FTN FORTRAN ERROR CALL
98	YES	NO	RETURN	YES	NO	YES	USER REQUESTED TRACEBACK

Table 3–2 (Cont.): Initial Error Control Bit Settings

Error			Continue		Permitted		
Number	Continue?	Count?	Type	Log?	Err =?	Return?	
100	NO	NO	FATAL	YES	NO	NO	DIRECTIVE: MISSING ARGUMENT(S)
101	NO	NO	FATAL	YES	NO	NO	DIRECTIVE: INVALID EVENT FLAG . . .
111	NO	NO	FATAL	YES	NO	NO	VIRTUAL ARRAY INITIALIZATION . . .
112	YES	YES	RETURN	YES	NO	YES	VIRTUAL ARRAY MAPPING ERROR

3.6 FORTRAN-77 Compiler Listing Format

mhere are three optional sections that you may include in a compiler listing file: the source program, the generated machine code, and the storage map. The source program and storage map are included in a list file by default. The generated machine language code is excluded by default. A description of each of these sections follows.

3.6.1 Source Listing

The source code of a compiled program is written into the source listing section of the compiler listing file in the same format as that in which the source code appears in the input file, except that the compiler adds internal sequence numbers to facilitate ease of reference. Comment lines and uncompiled debug statements, however, do not receive internal sequence numbers.

If the text editor you use generates line numbers, these numbers also appear in the source listing. They appear in the left margin, with the compiler-generated sequence numbers shifted to the right. Diagnostic messages always refer to the compiler-generated sequence numbers.

3.6.2 Generated Code Listing

The generated code listing section of the compiler listing file contains symbolic representations of object code generated by the compiler. These representations are similar to a MACRO-11 source listing, but they are not in a form that can be directly assembled by MACRO-11.

Labels that correspond to FORTRAN source labels are printed with an initial dot. For example, the source label "300" would appear in a generated code listing as ".300". Not all labels appearing in a source program necessarily appear in the corresponding generated code listing. In particular, labels not referenced in a source program are ignored by the compiler and are not used in resulting generated code.

References to variables and arrays defined in a source program are shown in the corresponding generated code listing by their FORTRAN names.

PDP-11 general registers 0 through 5 are represented in a generated code listing by R0 through R5, general register 6 is represented by SP (for Stack Pointer), and general register 7 is represented by PC (for Program Counter); the floating point registers are represented by F0 through F5. These representations are the conventional PDP-11 register names and are used despite the fact that you can also use these names as FORTRAN variable names.

In some cases, the compiler generates labels for its own use. These labels are shown in a generated code listing as "L$xxxx", where "xxxx" is a unique symbol for each label within a program unit.

Addresses for other than labels, registers, and variables are represented by the name of the program section plus the offset within that section. Program section names used by the compiler are summarized in Section 3.3.1. Changes from one program section to another are shown as .PSECT lines. The left column of a listing shows the offset within the current section to which the remainder of the line applies.

All numbers are in octal radix.

The first line of a generated code listing contains a .TITLE directive; for SUBROUTINE and FUNCTION subprograms, the title is the same as the subprogram name. If a PROGRAM statement is used in a main program, the name in that statement is used as the title; otherwise, the title .MAIN. is used. If a name is included in a BLOCKDATA statement, this name is used for the title; otherwise, the title .DATA. is used.

The second line of a generated code listing contains an .IDENT directive in which the date of the compilation is represented.

The lines that follow the second line describe the contents of storage initialized for FORMAT statements, DATA statements, constants, subprogram call argument lists, and so forth.

Machine instructions are represented in a generated code listing with MACRO-11 mnemonics and syntax.

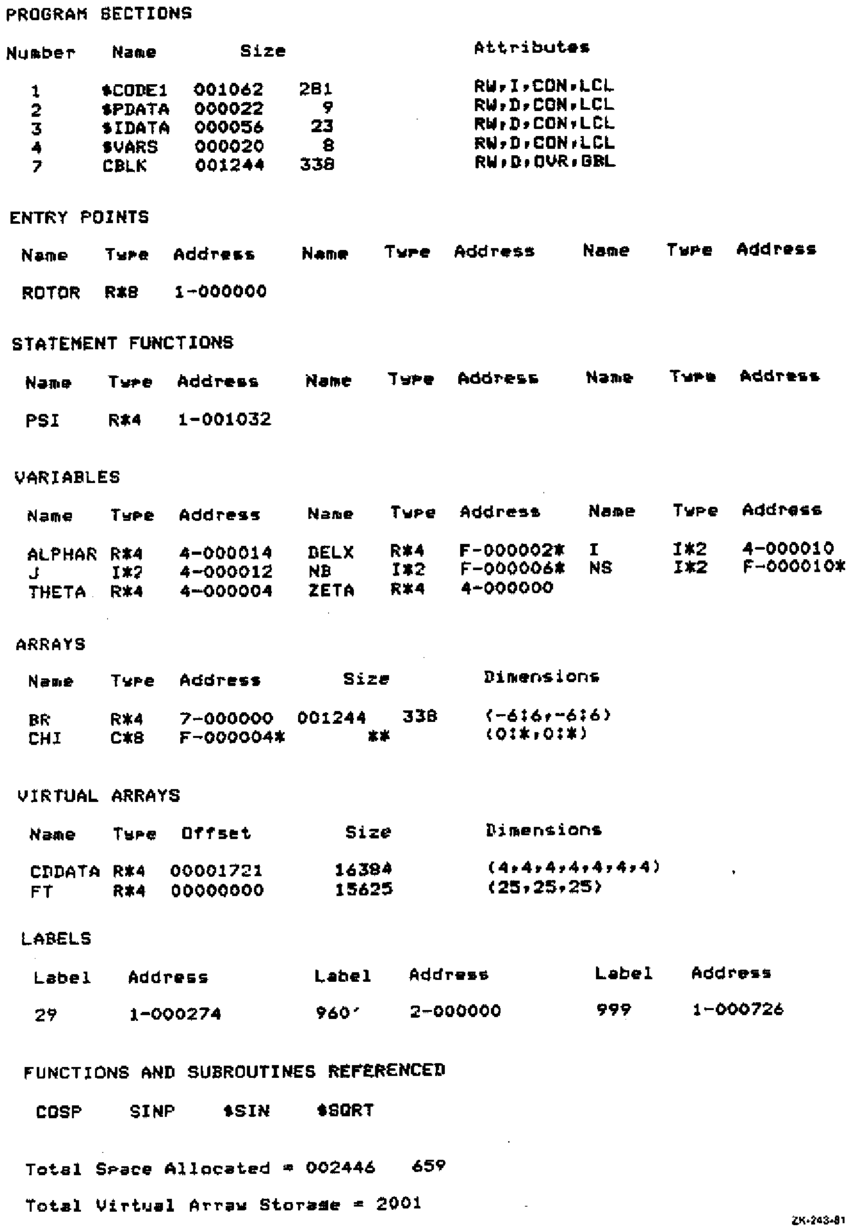
3.6.3 Storage Map Listing

The storage map contains summaries of the following:

- Program sections
- Entry points
- Variables
- Arrays
- Virtual arrays
- Labels
- Functions and subroutines referenced
- Total memory allocated

Figure 3-1 illustrates a typical storage map listing.

Figure 3–1: Storage Map Example



```
PROGRAM SECTIONS

Number    Name        Size                    Attributes

  1      $CODE1   001062   281                RW,I,CON,LCL
  2      $PDATA   000022     9                RW,D,CON,LCL
  3      $IDATA   000056    23                RW,D,CON,LCL
  4      $VARS    000020     8                RW,D,CON,LCL
  7      CBLK     001244   338                RW,D,OVR,GBL

ENTRY POINTS

 Name    Type   Address      Name    Type   Address      Name    Type   Address

 ROTOR   R*8    1-000000

STATEMENT FUNCTIONS

 Name    Type   Address      Name    Type   Address      Name    Type   Address

 PSI     R*4    1-001032

VARIABLES

 Name    Type   Address      Name    Type   Address      Name    Type   Address

 ALPHAR  R*4    4-000014     DELX    R*4    F-000002*    I       I*2    4-000010
 J       I*2    4-000012     NB      I*2    F-000006*    NS      I*2    F-000010*
 THETA   R*4    4-000004     ZETA    R*4    4-000000

ARRAYS

 Name    Type   Address         Size          Dimensions

 BR      R*4    7-000000   001244    338      (-6:6,-6:6)
 CHI     C*8    F-000004*        **           (0:*,0:*)

VIRTUAL ARRAYS

 Name    Type   Offset          Size          Dimensions

 CDDATA  R*4   00001721        16384          (4,4,4,4,4,4,4)
 FT      R*4   00000000        15625          (25,25,25)

LABELS

 Label    Address             Label    Address             Label    Address

 29       1-000274            960'     2-000000            999      1-000726

FUNCTIONS AND SUBROUTINES REFERENCED

 COSP     SINP     $SIN     $SQRT

Total Space Allocated = 002446   659

Total Virtual Array Storage = 2001
```

ZK-243-81

In each of the following descriptions, when a size is given, this size is printed as octal bytes followed by decimal words (except for virtual arrays). For example:

```
000006      3
```

A data address is given as a program section number followed by the octal offset from the beginning of that program section.

For example, in the data address that follows, 1 is the program section number and 000626 is the offset (in octal) from the beginning of program section 1:

```
1-000626
```

A dummy argument is represented with an F instead of a program section number, and the offset is the offset from the argument pointer (R5).

The symbol * following an address field specifies that the program section number (or F), plus the offset, points to the address of the data rather than to the data itself.

The PROGRAM SECTIONS summary in a storage map contains information—one line per program section—about each of the program sections (PSECTs) generated by the compiler. Each line contains the number of the PSECT being summarized (used by most of the other summaries), the name of the section, the size of the section, and the attributes of the section. The size is shown twice: first, as the number of bytes in octal radix; and, second, as the number of words in decimal radix. See Section 3.3.1 for definitions of the section attributes.

The ENTRY POINTS summary contains a list of all declared entry points and their addresses. If the routine containing an entry point being listed is a function, the declared data type of this entry point is also included.

The VARIABLES summary contains a list of each simple variable, together with its data type and address.

The ARRAYS summary is the same as the VARIABLES summary, except that it supplies total array size information and detailed dimension information. If the array is an adjustable array or assumed-size array, the size of the array is specified as **, and each adjustable-dimension bound or assumed-size bound is specified as *.

The VIRTUAL ARRAYS summary is similar to the array summary. The address of a virtual array is shown as an offset, in 64 byte units, from the start of virtual array storage. The size is specified as the number of array elements, not the number of bytes.

The LABELS summary contains a list of all user-defined statement labels. If a label is marked with an apostrophe, the label is a format label. If the label address field contains **, the label is neither referenced nor used by the compiled code.

The FUNCTIONS AND SUBROUTINES REFERENCED summary contains a list of all external-routine references made by the source program.

If the text NO FPP INSTRUCTIONS GENERATED appears in the storage map, the FORTRAN–77 object module may not require the Floating Point Processor (FPP) for execution. See Section 5.4.1 for further information.

At the end of the above summaries, the total amount of memory allocated by the compilation for all program sections is printed as follows:

```
TOTAL SPACE ALLOCATED = 000502    161
```

If any virtual arrays are declared in the program, the total size in 64-byte units is given as follows:

```
TOTAL VIRTUAL ARRAY STORAGE = 632
```

If a summary section has no entries in a particular compilation, the summary headings are not printed.

3.7 Virtual Array Options

The VIRTUAL statement declares arrays that are assigned space outside a program's address space and that are manipulated through the VIRTUAL facility of PDP–11 FORTRAN–77. The VIRTUAL array facility allows arrays to be stored in large data areas that are accessed at high speed.

NOTE

> VIRTUAL arrays are supported only on operating systems that support the Memory Management Directives.

3.7.1 Limits on VIRTUAL Elements

VIRTUAL arrays are limited by the number of elements, not by the available storage. The maximum number of elements in a VIRTUAL array is 65535; there is no limit to the total size of the VIRTUAL arrays a program can access. The limit on elements is 65535 because PDP-11 FORTRAN-77 requires that the number of elements in an array not exceed the size of an unsigned INTEGER*2, which is 2**16-1.

The largest LOGICAL*1 VIRTUAL array is 32K words, or 65535 bytes; and the largest REAL*8 VIRTUAL array is 256K words, or 624280 bytes.

3.7.1.1 VIRTUAL and DIMENSION Statements

The syntax of the VIRTUAL statement is identical to that of the DIMENSION statement. However, there is a significant semantic difference between the two because of the limitations imposed on the DIMENSION statement. Local arrays declared by the DIMENSION statement are limited by the maximum memory available to the program. Section 3.7.2 demonstrates how to use the VIRTUAL feature in an existing program.

3.7.1.2 Memory Allocation for VIRTUAL Arrays

The Task Builder allocates a mapped array area below a task's header; this mapped array area is large enough to contain all the VIRTUAL arrays declared in a program.

A window of 4K words initially maps the first 4K words of the VIRTUAL array region. When a VIRTUAL array element lies outside the window, a Memory Management directive causes a remap operation to allow access.

3.7.1.3 Execution Time of Virtual Arrays

Using VIRTUAL arrays increases the execution time of a task because VIRTUAL array elements must be mapped to memory addresses. In general, the larger the VIRTUAL array, the greater the number of times mapping occurs; therefore, larger arrays generally take longer to execute than do smaller arrays.

The following example illustrates how using VIRTUAL arrays increases execution time:

```
      PARAMETER         N=3500
      VIRTUAL           A(N), B(N), C(N)
      DO 10 I= I,N
      A(I)=1234.
      B(I)=5678
10    C(I)=A(I)/B(I)
      STOP
      END
```

As declared in the program above, the VIRTUAL arrays A, B, and C are each too large (7000 words) to fit within a 4K-word window of memory. Each time an element outside the 4K-word window is accessed, remapping occurs. Thus, executing the DO loop requires 17,500 (3500*5) mappings. If only array C were VIRTUAL, however, then only two mappings would be needed to execute the loop.

You can also use the RSX-11 Version 4.0 Task Builder option /FM (fast mapping) to improve execution speed. For more information, see the *RSX-11M/M-PLUS Task Builder Manual.*

The operations in the program above can require as long as 14.1 seconds for execution on a PDP-11/60 running under RSX-11M, Version 3.2. By contrast, if arrays A, B, and C were declared with a DIMENSION statement in directly addressable memory, the same operations could require as little as 0.12 seconds in the same operating environment.

You can reduce the mapping of VIRTUAL arrays by breaking large arrays into smaller ones and/or by keeping consecutive accesses of array elements within the current 4K-word window.

3.7.2 Converting a Program to VIRTUAL Array Usage

You can convert an existing program to use VIRTUAL arrays simply by declaring the array with VIRTUAL statements instead of DIMENSION statements. In doing this, however, be sure to observe the usage restrictions for VIRTUAL arrays described in the *PDP-11 FORTRAN-77 Language Reference Manual.*

The following example illustrates general program conversion.

1. Identify the non-VIRTUAL arrays that are to be converted to VIRTUAL arrays.
2. Locate the DIMENSION and the type declaration statements in which these arrays are declared. Replace DIMENSION statements with equivalent VIRTUAL statements. Replace array-declarative type declaration statements with VIRTUAL statements to define the array dimension, and remove the dimensioning information from the type declaration statements.
3. Compile the program and observe all compilation errors. These errors occur where the syntax restrictions outlined in the *PDP-11 FORTRAN-77 Language Reference Manual* have been violated. In some cases, to use VIRTUAL arrays effectively you may need to reformulate the data structures.
4. Check the code to ensure that VIRTUAL array parameters are passed correctly to subprograms.

    a. If the argument list of a subprogram call includes an unsubscripted VIRTUAL array name, the argument list of the SUBROUTINE or FUNCTION statement must have an unsubscripted VIRTUAL array name in its corresponding dummy argument. This corresponding VIRTUAL array name establishes access to the VIRTUAL array for the subprogram. The declaration of the VIRTUAL array in the subprogram must be dimensionally compatible with the VIRTUAL declaration in the calling program. All changes to the VIRTUAL array that occurred during subprogram execution are retained when control returns to the calling program.

       When you pass entire arrays as subprogram parameters, be certain that the matching arguments are defined as both VIRTUAL or both non-VIRTUAL. Mismatches of array types are not detectable at either compilation or execution time, and the results are undefined.

b. If the argument list of a subprogram reference includes a reference to a VIRTUAL array element, the matching formal parameter in the SUBROUTINE or FUNCTION statement must be a non-VIRTUAL variable. Value assignments to the formal parameter occurring within the subprogram do not alter the stored value of the VIRTUAL array element in the calling program. To alter the value of that element, the calling program must include a separate assignment statement that references the VIRTUAL array element directly.

The following shows how to change non-VIRTUAL arrays to VIRTUAL arrays.

Consider a program containing two arrays, A and B.

```
      DIMENSION A(1000,20)
      INTEGER*2 B(1000)
      DATA B/1000*0/
      CALL ABC(A,B,1000,20)
      WRITE(2,*)(A(1,1),I=1,1000)
      END

      SUBROUTINE ABC(X,Y,N,M)
      DIMENSION X(N,M)
      INTEGER*2 Y(N)
      DO 10, I=1,N
10    X(I,1)=Y(I)
      RETURN
      END
```

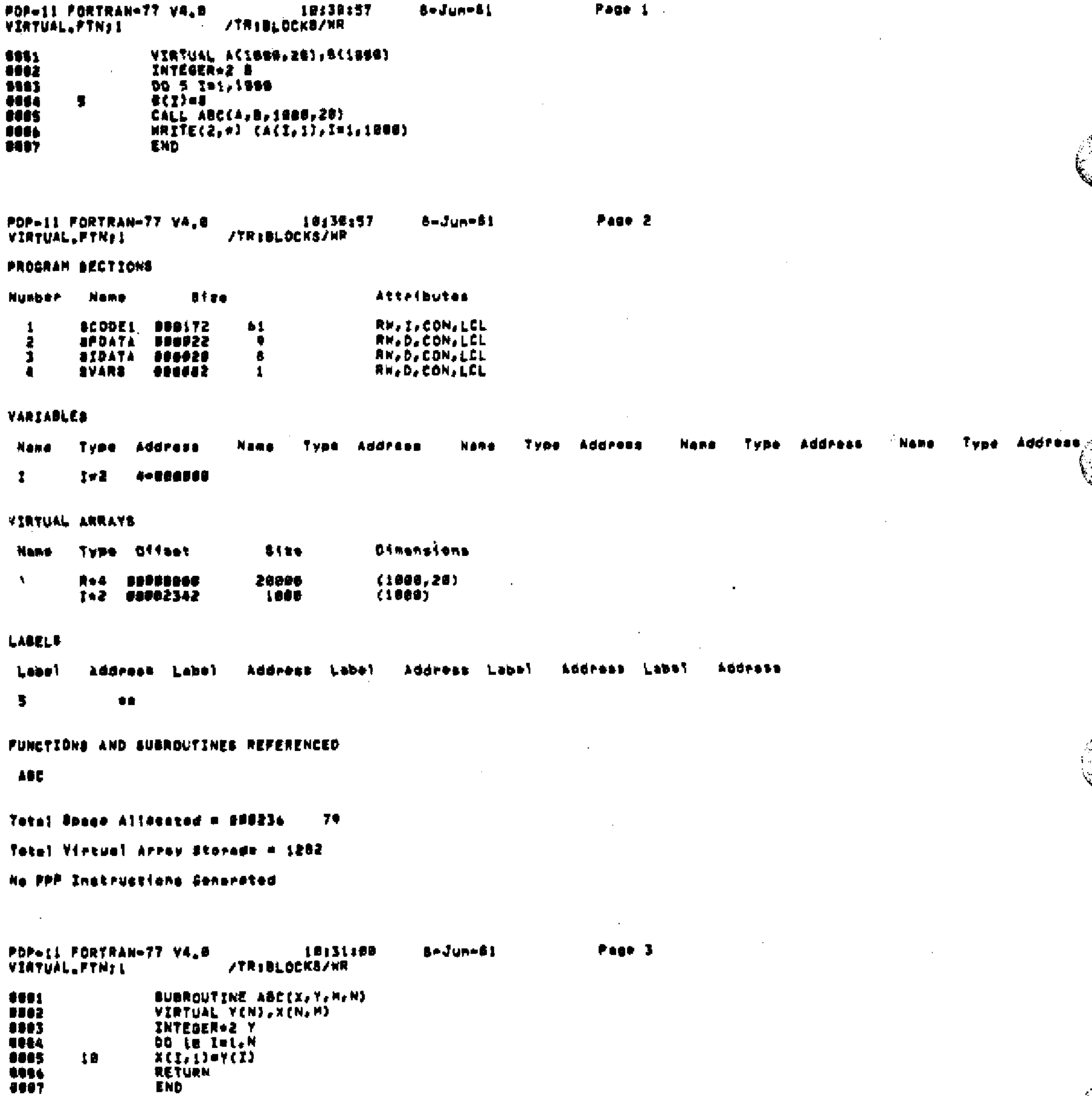
Array A is declared in a DIMENSION statement and is of the default data type. Therefore, substituting the keyword VIRTUAL for the keyword DIMENSION is sufficient for its conversion.

Note, however, that array B and its dimensions are declared in a type declaration statement (in the second line of the program).

To convert B into a VIRTUAL array, its declarator must be moved to a VIRTUAL statement; also, the variable B must remain in the type declaration statement, but without a dimension specification.

A and B are both passed to subroutine ABC as arrays, rather than array elements. Therefore, the associated subroutine parameters must also be converted to VIRTUAL arrays.

The following listing shows the program after the conversion is completed.



```
PDP-11 FORTRAN-77 V4.0            10:30:57      8-Jun-81            Page 1
VIRTUAL.FTN;1              /TR:BLOCKS/WR

0001               VIRTUAL A(1000,20),B(1000)
0002               INTEGER*2 B
0003               DO 5 I=1,1000
0004       5       B(I)=5
0005               CALL ABC(A,B,1000,20)
0006               WRITE(2,*) (A(I,1),I=1,1000)
0007               END


PDP-11 FORTRAN-77 V4.0            10:30:57      8-Jun-81            Page 2
VIRTUAL.FTN;1              /TR:BLOCKS/WR

PROGRAM SECTIONS

Number   Name        Size                 Attributes

  1      $CODE1   000172    61            RW,I,CON,LCL
  2      $PDATA   000022     9            RW,D,CON,LCL
  3      $IDATA   000020     8            RW,D,CON,LCL
  4      $VARS    000002     1            RW,D,CON,LCL

VARIABLES

 Name   Type  Address     Name   Type  Address     Name   Type  Address     Name   Type  Address     Name   Type  Address

 I      I*2   4-000000

VIRTUAL ARRAYS

 Name   Type  Offset         Size        Dimensions

 A      R*4   00000000       20000       (1000,20)
        I*2   00002342        1000       (1000)

LABELS

 Label    Address  Label    Address  Label    Address  Label    Address  Label    Address

 5          **

FUNCTIONS AND SUBROUTINES REFERENCED

 ABC

Total Space Allocated = 000236    79

Total Virtual Array Storage = 1202

No FPP Instructions Generated


PDP-11 FORTRAN-77 V4.0            10:31:00      8-Jun-81            Page 3
VIRTUAL.FTN;1              /TR:BLOCKS/WR

0001               SUBROUTINE ABC(X,Y,M,N)
0002               VIRTUAL Y(N),X(N,M)
0003               INTEGER*2 Y
0004               DO 10 I=1,N
0005      10       X(I,1)=Y(I)
0006               RETURN
0007               END
```

ZK-171-81

Chapter 4

PDP-11 FORTRAN-77 Implementation Concepts

This chapter discusses several of the fundamental design and implementation concepts of PDP–11 FORTRAN–77 that are different from those of other FORTRAN systems, or that are likely to be new to many FORTRAN programmers.

4.1 Intrinsic Functions

As it processes a program unit, the compiler determines (without any information about other program units that may be added later) whether a function referenced in the program unit is an intrinsic function (processor-defined) or a user-defined function. The compiler invokes an intrinsic function with a symbolic name, called an internal name, that is different from any name the user can define. For example, the intrinsic real-valued sine function is invoked by the compiler with the internal name $SIN.

In general, an internal name is a FORTRAN name with a dollar sign prefixed. Where the FORTRAN name is six characters long, a 5-character contraction is combined with the dollar sign. A complete list of the intrinsic names and their corresponding internal names appears in Table 4–1.

Using the IMPLICIT statement to change the default data type rules has no effect on the data type of intrinsic functions.

4.1.1 Using EXTERNAL and INTRINSIC Statements

The EXTERNAL statement identifies symbolic names as user-supplied functions and subroutines. The INTRINSIC statement identifies symbolic names as system-supplied functions or subroutines. For example, the statement

EXTERNAL INVERT

identifies a subroutine named INVERT as user-supplied, and

INTRINSIC ABS

identifies a function named ABS as system-supplied.

Once a symbolic name has been identified in an EXTERNAL statement, it is no longer available in the same program unit for use in an INTRINSIC statement.

Refer to Appendix E for information on the compatibility of the EXTERNAL statement with PDP–11 FORTRAN–77 and PDP–11 FORTRAN IV-PLUS programs.

4.1.2 Generic Function References

A generic function is similar to an intrinsic function, but instead of being a single function it is a set of similar functions called specific functions. The specific functions in a generic set differ from each other only in that each function manipulates data of one specific type. For example, SIN( ) is a generic function that includes the specific functions SIN, DSIN, and CSIN, where SIN manipulates real data, DSIN double-precision data, and CSIN complex data. The data type of the argument in a generic reference determines which specific function is actually invoked. For example, SIN(X) invokes SIN if X is real and DSIN if X is double precision. The compiler makes a separate determination of the specific function to be referenced each time it encounters the same generic reference.

Those intrinsic functions that can be referenced by generic references are listed in Table 4–1 under the heading "Generic Name." Many generic function names are also intrinsic function names. However, in a few cases (for example, the generic function name MIN), the generic function name is not an intrinsic function name.

Table 4–1: Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Square Root[2]	1	SQRT	SQRT	Real	Real
			DSQRT	Double	Double
a(1/2)			CSQRT	Complex	
					Complex
Natural Logarithm[3]	1	LOG	ALOG	Real	Real
			DLOG	Double	Double
log(e)a			CLOG	Complex	
					Complex
Common Logarithm[3]	1	LOG10	ALOG10	Real	Real
				Double	Double
log(10)a			DLOG		
Exponential	1	EXP	EXP	Real	Real
			DEXP	Double	Double
e(a)			CEXP	Complex	
					Complex
Sine[4]	1	SIN	SIN	Real	Real
			DSIN	Double	Double
sin a			CSIN	Complex	
					Complex
Cosine[4]	1	COS	COS	Real	Real
			DCOS	Double	Double
cos a			CCOS	Complex	
					Complex
Tangent[4]	1	TAN	TAN	Real	Real
			DTAN	Double	Double
tan a					

[2] The argument of SQRT and DSQRT must be greater than or equal to 0. The result of CSQRT is the principal value with the real part greater than or equal to 0. When the real part is 0, the result is the principal value with the imaginary part greater than or equal to 0.

[3] The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than 0. The argument of CLOG must not be (0.,0.).

[4] The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. The argument is treated as modulo 2•pi.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Arc Sine[5,6]	1	ASIN	ASIN DASIN	Real Double	Real Double
arc sine a					
Arc Cosine[5,6]	1	ACOS	ACOS DACOS	Real Double	Real Double
arc cos a					
Arc Tangent[6]	1	ATAN	ATAN DATAN	Real Double	Real Double
arc tan a					
Arc Tangent[6,7]	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
arc tan a(1) /a(2)					
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
sinh a					
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Cosh a					
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Tanh a					

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

[5] The absolute value of the argument of ASIN, DASIN, ACOS, and DACOS must be less than or equal to 1.

[6] The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

[7] The result of ATAN2 and DATAN2 is 0 or positive when a(2) is less than or equal to 0. The result is undefined if both arguments are 0.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Absolute value[8]	1	ABS	ABS	Real	Real
			DABS	Double	Double
[a]			CABS	Complex	
			IIABS	INTEGER*2	Real
			JIABS		INTEGER*2
				INTEGER*4	
					INTEGER*4
		IABS	IIABS	INTEGER*2	INTEGER*2
			JIABS		
				INTEGER*4	INTEGER*4
Truncation[9]	1	INT	IINT	Real	INTEGER*2
			JINT	Real	
[a]			IIDINT	Double	INTEGER*4
			JIDINT	Double	
					INTEGER*2
					INTEGER*4
		IDINT	IIDINT	Double	INTEGER*2
			JIDINT	Double	
					INTEGER*4
		AINT	AINT	Real	Real
			DINT	Double	Double
Nearest Integer[9]	1	NINT	ININT	Real	INTEGER*2
			JNINT	Real	
[a+.5*sign(a)]			IIDNNT	Double	INTEGER*4
				Double	
			JIDNNT		INTEGER*2
					INTEGER*4
		IDNINT	IIDNNT	Double	INTEGER*2
				Double	
			JIDNNT		INTEGER*4

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

[8] The absolute value of a complex number, (X,Y), is the real value: (X(2)+Y(2))(1/2)

[9] [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [-5.7] equals -5.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
		ANINT	ANINT	Real	Real
			DNINT	Double	Double
Fix[10] (real-to-integer conversion)	1	IFIX	IIFIX	Real	INTEGER*2
			JIFXI	Real	
					INTEGER*4
Float[10] (integer-to-real conversion)	1	FLOAT	FLOATI	INTEGER*2	Real
					Real
			FLOATJ	INTEGER*4	
Double Precision Float[10] (integer-to-double conversion)	1	DFLOAT	DFLOTI	INTEGER*2	Double
			DFLOTJ	INTEGER*4	Double
Conversion to Single Precision[10]	1	SNGL	-	Real	Real
			SNGL	Double	Real
			FLOATI	iNTEGER*2	Real
					Real
			FLOATJ	INTEGER*4	
Conversion to Double Precision[10]	1	DBLE	DBLE	Real	Double
			-	Double	
			-	Complex	Double
			DFLOTI	INTEGER*2	
					Double
			DFLOTJ	INTEGER*4	
					Double
					Double
Real Part of Complex or Conversion to Single Precision[10]	1	REAL	REAL	Complex	Real
			FLOATI	INTEGER*2	Real
					Real
			FLOATJ	INTEGER*4	Real
					Real
			SNGL	Real	
			SNGL	Double	

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

[10] Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function SNGL with a real argument and the function DBLE with a double precision argument return the value of the argument without conversion.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Imaginary Part of Complex	1	-	AIMAG	Complex	Real
Conversion to	1,2	CMPLX	-	INTEGER*2	Complex
Complex	1,2		-		
or	1,2		-	INTEGER*4	Complex
Complex from Two	1,2		CMPLX		
Arguments[11]	1,2		-	Real	Complex
	1		-	Real	
				Double	Complex
				Complex	
					Complex
					Complex
Complex Conjugate	1	-	CONJG	Complex	Complex
(if a= (X,Y)					
CONJG (a)=(X,Y)					
Double Product of Reals	2	-	DPROD	Real	Double
a(1)*a(2)					
Maximum	n	MAX	AMAX1	Real	Real
				Double	Double
max (a(1,),a(2), . . . a			DMAX1	INTEGER*2	
(n)) (returns the max-					INTEGER*2
imum value from			IMAX0	INTEGER*4	
among the argument			JMAX0		INTEGER*4
list; there must be at					
least two arguments)					
		MAX0	IMAX0	INTEGER*2	
			JMAXO		
				INTEGER*4	

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

[11] When CMPLX has only one argument, this argument is converted into the real part of a complex value, and zero is assigned to the imaginary part. When CMPLX has two arguments, the first argument is converted to the real part of a complex value, the second to the imaginary part.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
		MAX1	IMAX1	Real	INTEGER*2
			JMAX1	Real	
					INTEGER*4
		AMXA0	AIMAX0	INTEGER*2	Real
					Real
			AJMAX0	INTEGER*4	
Minimum	n	MIN	AMIN1	Real	Real
			DMIN1	Double	Double
min(a(1),a(2),..a(n)) (returns the minimum value among the argument list; there must be at least two arguments)			IMIN0	INTEGER*2	
			JMIN0		INTEGER*2
				INTEGER*4	
		MIN0	IMIN0		INTEGER*4
			JMINO		
				INTEGER*2	
		MIN1	IMIN0		INTEGER*2
			JMIN0	INTEGER*4	
					INTEGER*4
		AMIN0	AIMIN0		
				Real	
				Real	INTEGER*2
			AJMIN0		
				INTEGER*2	INTEGER*4
				INTEGER*4	
					Real
					Real
Positive Difference	2	DIM	DIM	Real	Real
			DDIM	Double	Double
a(1)-(min(a(1),a(2))) returns the first argument minus the minimum of the two arguments)			IIDIM	INTEGER*2	
			JIDIM		INTEGER*2
				INTEGER*4	
		IDIM	IIDIM		INTEGER*4
			JIDIM		
				INTEGER*2	
					INTEGER*2
				INTEGER*4	
					INTEGER*4

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Remainder	2	MOD	AMOD	Real	Real
			DMOD	Double	Double
a(1)-a(2)*[a(1)/a(2)] (returns the remainder when the first argument is divided by the second)			IMOD	INTEGER*2	INTEGER*2
			JMOD	INTEGER*4	INTEGER*4
Transfer of Sign	2	SIGN	SIGN	Real	Real
			DSIGN	Double	Double
a(1) *Sign a(2)			IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
		ISIGN	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND	INTEGER*2	
			JIAND	INTEGER*4	
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR	INTEGER*2	
			JIOR	INTEGER*4	
Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR	INTEGER*2	INTEGER*2
			JIEOR	INTEGER*4	INTEGER*4
Bitwise Complement (complements each bit)	1	NOT	INOT	INTEGER*2	INTEGER*2
			JNOT	INTEGER*4	INTEGER*4
Bitwise Shift (a(1) logically shifted left a(2) bits)	2	ISHFT	IISHFT	INTEGER*2	INTEGER*2
			JISHFT	INTEGER*4	INTEGER*4

[1] *See Section 4.2.4 for definitions of "I" and "J" forms.*

Table 4–1 (Cont.): Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name[1]	Type of Argument	Type of Result
Random Number[12](returns the next number from a sequence of pseudorandom numbers of	1	-	RAN	INTEGER*4	Real
uniform distribution	2	-	RAN		Real
over the range 0 to 1)				INTEGER*2	
Length (returns length of the character expression)	1	-	LEN	Character	INTEGER*2
Index (C(1),C(2)) (returns the position of the substring c(2) in the character expression c(1))	2	-	INDEX	Character	INTEGER*2
ASCII Value (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	-	ICHAR	Character	INTEGER*2
Character relationals	2	-	LLT	Character	Logical*2
(ASCII collating se-	2	-	LLE	Character	
quence)	2	-	LGT	Character	Logical*2
	2	-	LGE	Character	
					Logical*2
					Logical*2

[1] See Section 4.2.4 for definitions of "I" and "J" forms.

[12] The argument for this function must be an integer variable or integer array element. The argument should initially be set to 0. The RAN function stores a value in the argument that it later uses to calculate the next random number. Resetting the argument to 0 regenerates the sequence. Alternate starting values generate different random-number sequences.

4.2 INTEGER*2 and INTEGER*4

PDP–11 FORTRAN–77 provides two integer data types: INTEGER*4, for purposes of high precision; and INTEGER*2, for purposes of efficiency. INTEGER*4 operations are performed to 32 bits of significance; however, because these operations require more instructions and storage than INTEGER*2 operations, they are less efficient in terms of both time and memory.

To encourage efficiency, the FORTRAN–77 compiler assumes all integer variables to be of INTEGER*2 types unless you explicitly declare them to be INTEGER*4 within a program, or unless you set the /I4 compiler switch (see Section 1.2.4).

When in INTEGER*4 mode, the compiler treats all integer (and logical) variables as INTEGER*4 (and LOGICAL*4) types unless you explicitly declare them otherwise within a program.

4.2.1 Representation and Relationship of INTEGER*2 and INTEGER*4 Values

INTEGER*2 values are stored as two's complement binary numbers in one word of storage. INTEGER*4 values are represented in two's complement binary form in two words of storage: the first word (lower address) contains the low-order part of the value, and the second word (higher address) contains the high-order part of the value (including sign).

An INTEGER*2 value is, then, a subset of an INTEGER*4 value. Therefore, the address of an INTEGER*4 value within the range -32768 to +32767 can be treated as the address of an INTEGER*2 value; and conversion from INTEGER*4 to INTEGER*2 (without overflow checking) consists of ignoring the high-order word of the INTEGER*4 value. (In situations where you can determine at compile time that the results will not be affected, you can generate INTEGER*2 code to perform INTEGER*4 operations.)

The FORTRAN rules state that corresponding actual and dummy arguments must agree in type. In the following example, however, if the compiler supplies an INTEGER*2 constant as the actual argument, SUB executes correctly even if its dummy argument is of INTEGER*4 data type:

```
CALL SUB(2)
```

4.2.2 Integer Constant Typing

In general, typing integer constants as either INTEGER*2 or INTEGER*4 is based on the magnitude of the constant; and in most contexts, INTEGER*2 and INTEGER*4 variables and integer constants may be freely mixed. However, the programmer is responsible for ensuring that integer overflow conditions that might adversely affect the program do not occur. Consider 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, overflows the valid INTEGER*2 range, and the resulting bit pattern represents -32766, a valid INTEGER*2 value that is converted to INTEGER*4 type and assigned to J. This overflow is not detected.

Compare the above example with the following apparently equivalent program, which produces an entirely different, and logically correct, result:

```
INTEGER*4   J
PARAMETER   I = 32767
J = I + 3
```

In this example, the compiler adds the constant 3 and the parameter constant 32767 and produces a resulting constant of 32770. The compiler recognizes this constant as an INTEGER*4 value and assigns it to J.

4.2.3 Octal Constant Typing

Octal constants can take either of two forms:

```
'C1 C2 C3...Cn'

"C1 C2 C3...Cn
```

Octal constants of the form 'C1 C2 C3 . . . Cn' O are typeless numeric constants that assume data types on the basis of the way they are used. See the *PDP-11 FORTRAN-77 Language Reference Manual* for the rules on the typing of octal constants of this form.

Octal constants of the form C1 C2 C3 . . . Cn, however, are typed as either INTEGER*2 or INTEGER*4, and are typed on the basis of the magnitude of the constant.

An octal constant of the form C(1) C(2) C(3) . . . C(n) is typed as INTEGER*2 if bits 16 through 31 of the value are the same as bit 15; otherwise, it is typed as INTEGER*4. Because octal constants are treated as unsigned values, they are interpreted as positive values unless bit 31 is set. The octal constants 100000 through 177777 are typed as INTEGER*4 and interpreted as the decimal values 32768 through 65535, rather than as the negative signed decimal values -32768 through -1.

Because octal constants are positive values, you must take care when you compare octal constants with negative signed INTEGER*2 values.

Consider the following example:

```
INTEGER*2 I
IF (I .EQ. "105132) STOP
```

The comparison made here always results in an inequality (and the STOP statement is not executed). The reason for this is that the INTEGER*2 value of I is converted to INTEGER*4 before the comparison (to conform with the type of 105132); therefore, whenever I contains the bit pattern 105132, this pattern will be interpreted after conversion as the negative decimal value -30118.

The above example is equivalent to:

```
INTEGER*2 I
IF (I .EQ. 35418) STOP
```

If INTEGER*2 values must be compared with octal constants of the form 1xxxxx, the octal constant should be assigned to an INTEGER*2 temporary. An INTEGER*2 temporary could be used in our example as follows:

```
INTEGER*2 I, ICONST
DATA ICONST/"105312/
IF (I .EQ. ICONST) STOP
```

4.2.4 Integer-Valued Intrinsic Functions

A number of the intrinsic functions provided by FORTRAN-77 (for example, IFIX) produce integer results from real or double-precision arguments. These intrinsic functions are called "result generic" functions. Because the compiler operates in two different modes, INTEGER*2 mode and INTEGER*4 mode, the system provides two internal versions of each of these integer-producing functions: an INTEGER*2 version and an INTEGER*4 version. The compiler selects the proper version on the basis of the current compiler mode setting rather than—as it does for the other intrinsic functions—on the basis of the data type of arguments in the function reference.

In some cases, you may need to use the version of an integer intrinsic function that is the opposite of the one that would be invoked under the current compiler mode setting. For example, a program that predominantly uses INTEGER*2 values may at some point need to get an INTEGER*4 result from a intrinsic function. To satisfy this need, the system provides an additional pair of intrinsic function names that can reference the two internal versions of each integer-producing intrinsic function no matter what the current compiler mode setting may be. By convention, these additional names are created by prefixing I and J to the intrinsic function name. For example, I is prefixed to IFIX to create the INTEGER*2 version of this function name, and J is prefixed to create the INTEGER*4 version. IIFIX references the INTEGER*2 internal function $IFIX, and JIFIX references the INTEGER*4 internal function $JFIX.

The complete set of names and corresponding internal routines is shown in Table 4-1 (in Section 4.1).

4.2.5 Implementation-Dependent Integer Typing

The FORTRAN-77 compiler performs a number of integer-typing optimizations by taking advantage of certain properties of the PDP-11 and/or the operating system. These optimizations are generally transparent to a FORTRAN user and include the following:

- Array addressing calculations

  Because the entire virtual address space of the PDP-11 can be represented in one word, array bounds expressions and array subscript expressions are always converted to INTEGER*2 before being used in an array address calculation. Therefore, even when the compiler is operating in /I4 mode, the code generated for array addressing is performed with INTEGER*2 operations.

- Input/output logical unit numbers

  Because logical unit numbers can always be represented by a 1-word integer, the compiler converts all unit numbers to INTEGER*2 when producing calls to the I/O section of the OTS.

- Direct access record numbers

  For easy implementation, and to provide programs that predominantly use 1-word integers the capability of using very large files, all direct access record numbers are processed as INTEGER*4 values.

4.3 BYTE (LOGICAL*1) Data Type

FORTRAN–77 provides the byte data type (BYTE) to take advantage of the byte-processing capabilities of the PDP–11. Although LOGICAL*1 is a synonym for BYTE, a BYTE value is actually a signed integer. In addition to storing small integers, the byte data type is used for keyed access to indexed files and for storing and manipulating Hollerith information.

In general, when data of two different types are used in a binary operation, the lower-ranked type is converted, before any computations, to the higher-ranked type. However, in the case of a byte variable and an integer constant that can be represented as a byte variable, the integer constant is treated as a byte constant; therefore, the result of the operation is of type byte rather than of type integer, as it would be under the more general convention. The overflow possibilities under this convention, however, are similar to those in Section 4.2.2 for mixed INTEGER*2 and INTEGER*4 variables and constants.

4.4 Iteration Count Model for DO Loops

FORTRAN-77 provides an extended form of the DO statement. This statement has the following features:

- The control variable may be an INTEGER*2, INTEGER*4, REAL, or DOUBLE PRECISION variable.
- The initial value, step size, and final value of the control variable can be represented by any expressions whose resulting types are INTEGER*2, INTEGER*4, REAL, or DOUBLE PRECISION.
- The number of times the loop is executed (the iteration count) is determined when the DO statement is initialized and is not reevaluated during successive executions of the loop. Thus, the number of times the loop is executed is not affected by changing the values of the parameter variables used in the DO statement.

4.4.1 Cautions Concerning Program Interchange

Three common practices associated with the use of DO statements on other FORTRAN systems may not have the intended effects when used with FORTRAN-77. These are as follows:

- 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 cause an arbitrarily long loop that is terminated by a conditional control transfer within the loop results in zero iterations of the loop body. A zero step size may result in an infinite loop at run time.

4.4.2 Iteration Count Computation

Given the following generic DO statement:

```
DO label V=m1,m2,m3
```

(where m1, m2, and m3 are any expressions), the iteration count is computed as follows:

```
count= MAX(INT(m2-m1+m3)/m3,0)
```

This computation does the following:

- Provides that the body of the DO loop will be executed zero times if the iteration count given by the above formula is zero. (Under the /NOF77 switch, the loop is executed one time if the iteration count is zero.)
- Permits the step size (m3) to be negative or positive, but not zero
- Gives a well-defined and predictable value of an iteration count that results from any combination of values of the allowed result types

Note that overflow of INTEGER*2 control variables is not detected and can result in an infinite loop at run time. Consider the following program unit:

```
 DO 10 I=1,32767
         .
         .
         .

    10 CONTINUE
```

This program unit always results in an infinite loop when I is of INTEGER*2 type. See Section 4.2.2 for more information on integer overflow conditions.

You should also be aware that the effects of round-off error inherent in any floating-point computation, when real or double-precision values are used, may cause the count to be greater than, or less than, desired.

Under certain conditions, it is not necessary to actually compute the iteration count to obtain the required number of iterations; if all the parameters in an iteration computation are of type integer, and the step size is a constant (so that the sign of the increment value is known), the FORTRAN-77 compiler generates the necessary code to compare the control variable directly with the final value to control the number of iterations of the loop.

4.5 Using EQUIVALENCE with Mixed Data Types

You can predict the effects of EQUIVALENCE statements involving variables and/or arrays of mixed type when you consider the actual storage (in bytes) of each type of variable involved.

Example 4–1 illustrates the relationships that result when an EQUIVALENCE statement uses byte, integer, real, and complex elements.

Character data must not be equivalenced to data of any type other than character, BYTE, or LOGICAL*1.

Example 4–1: EQUIVALENCE Using Mixed Data Types

```
BYTE B (0:9)
COMPLEX C(4)
REAL R(3)
INTEGER*2 I(3)
EQUIVALENCE (C(2),R(3),I),(I(3),B(9))

Address       Storage Alignment

n        C(1) R(1)
n+1        .    .
n+2        .    .
n+3        .    .         B(0)
n+4        .  R(2)        B(1)
n+5        .    .         B(2)
n+6        .    .         B(3)
n+7        .    .         B(4)
n+8      C(2) R(3)   I(1) B(5)
n+9        .    .       . B(6)
n+10       .    .    I(2) B(7)
n+11       .    .       . B(8)
n+12       .         I(3) B(9)
n+13       .            .
n+14       .
n+15       .
n+16     C(3)
```

4.6 Equivalence, BYTE Data, and Storage Alignment

The PDP-11 hardware requires that storage for all data elements except byte elements begin at an even address. This requirement can be satisfied in all except the following two cases:

- Equivalence relationships involving byte elements and nonbyte elements can make it logically impossible to allocate variables in a manner that satisfies the even-byte alignment constraint for all elements involved in an equivalence. An example of such an equivalence relationship is as follows:

```
BYTE B(2)
INTEGER*2 I,J
EQUIVALENCE (B(1),I),(B(2),J)
```

- Using a COMMON block in more than one program unit constitutes an implied relationship of equivalence among the sets of elements declared in that block. If a strict interpretation of the sequence of variable allocations causes a nonbyte variable to start at an odd address, a compiler adjustment is not made because it could destroy alignment properties expected in another program unit.

The compiler begins allocating each common block, and each group of equivalenced variables that are not in common, at an even address. If an allocation results in an element not of type byte being stored beginning at an odd address, an error message is produced. If this happens, to avoid fatal errors during execution, you must modify the common and/or EQUIVALENCE statements to eliminate the odd-byte addressing.

Variables and arrays not in common and not used in EQUIVALENCE statements are always correctly aligned.

4.7 ENTRY Statement Arguments

The FORTRAN-77 implementation of argument association in ENTRY statements varies from that of some other FORTRAN systems.

As mentioned in Chapter 3, FORTRAN-77 uses the call-by-reference method of passing arguments to called procedures. Some other FORTRAN implementations use the call-by-value/result method. This difference in approach is important to keep in mind when you reference dummy arguments in ENTRY statements.

Although standard FORTRAN allows you to use the same dummy arguments in different ENTRY statements, it allows you to reference only those dummy arguments that are defined for the ENTRY point being called. For example, given the subprogram unit

```
SUBROUTINE SUB1(X,Y,Z)
    .
    .
    .
ENTRY ENT1(X,A)
    .
    .
    .
ENTRY ENT2(B,Z,Y)
```

you can make the following references:

```
CALL     Valid References

SUB1       X    Y    Z
ENT1       X    A
ENT2       B    Z    Y
```

FORTRAN implementations that use the call-by-value/result method, however, permit you to reference dummy arguments that are not defined in the ENTRY statement being called. For example, consider the following device for initializing dummy variables for subsequent referencing:

```
SUBROUTINE INIT(A,B,C)
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

You can use this nonstandard device in call-by-value/result implementations because a separate internal variable is allocated for each dummy argument in the called procedure. When the procedure is called, each scalar actual-argument value is assigned to the corresponding internal variable, and these internal variables are then used whenever there is a reference to a dummy argument within the procedure. On return from the procedure, modified dummy arguments are copied back to the corresponding actual-argument variables.

When an entry point is referenced, all the dummy arguments of the entry point are defined with the values of the corresponding actual arguments and can be referenced on subsequent calls to the subprogram. However, you should avoid such subsequent referencings in programs that are to be compiled under FORTRAN-77, as they will not have the intended effect and will produce programs that are not transportable to other systems that use the call-by-reference method.

FORTRAN–77 creates associations between dummy and actual arguments by passing the address of each actual argument to the called procedure. Each subsequent 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. Therefore, to perform the kind of nonstandard references shown in the previous example, the subprogram would have to copy the values of the dummy arguments to other variables. For example, if subroutine INIT is rewritten as follows, it will work on FORTRAN–77 as well as on systems that use the call-by-value/result method:

```
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
```

Chapter 5

PDP–11 FORTRAN–77 Programming Considerations

This chapter discusses techniques for writing effective FORTRAN–77 programs. Topics discussed are as follows:

- Efficient use of program statements and data types
- Compiler optimizations
- Program size and speed considerations
- Optional OTS capabilities
- RMS–11 and FCS link and run-time considerations

5.1 Creating Efficient Source Programs

The following sections discuss the use of the PARAMETER, INCLUDE, OPEN, and CLOSE statements in relation to writing efficient source programs; they also discuss the efficient use of the INTEGER*2 and INTEGER*4 data types.

5.1.1 PARAMETER Statement

The PARAMETER statement provides a way for you to write programs containing easily modified parameters, such as array bounds and iteration counts, without losing the efficiency of using constant expressions to manipulate these parameters. Because the FORTRAN-77 compiler can optimize constants more efficiently than it can optimize variables (see Section 5.2.2), programs that use PARAMETER statements are generally more efficient than programs that initialize parameters with DATA or assignment statements. For example, the first program fragment below compiles into more efficient code than the second or third:

```
(1)         PARAMETER(M=50,N=100)
            DIMENSION X(M),Y(N)
            DO 5, I=1,M
            DO 5, J=1,N
      5     X(I) = X(I)*Y(J) + X(M)*Y(N)

(2)         DIMENSION X(50),Y(100)
            DATA M,N/50,100/
            DO 5, I=1,M
            DO 5, J=1,N
      5     X(I) = X(I)*Y(J) + X(M)*Y(N)

(3)         DIMENSION X(50),Y(100)
            M = 50
            N = 100
            DO 5, I=1,M
            DO 5, J=1,N
      5     X(I) = X(I)*Y(J) + X(M)*Y(N)
```

5.1.2 INCLUDE Statement

The INCLUDE statement provides a way for you to eliminate duplication of source code and to facilitate program maintenance. Because of the availability of the INCLUDE statement, you can create and maintain a separate file for a section of program text used by several different program units, and then include this text in the individual program units at compile time. For example, rather than duplicate the specification for a common block referenced by several program units, you can write the specification a single time in a separate file; then each program unit referencing the common block merely executes an INCLUDE statement to incorporate the specification into the unit. In addition to increasing programming efficiency, using the INCLUDE statement fosters reliability, modular programming, and ease of maintenance.

The following example shows the use of the INCLUDE statement.

The file COMMON.FTN defines the size of the blank common block and the size of the arrays X,Y, and Z.

Main Program File

```
      INCLUDE 'COMMON.FTN'
      DIMENSION Z(M)
      CALL CUBE
      DO 5 I=1,M
5     Z(I)=X(I)+SQRT(Y(I))
          .
          .
          .
      SUBROUTINE CUBE
      INCLUDE 'COMMON.FTN'
      DO 10 I=1,M
10    X(I)=Y(I)**3
      RETURN
      END
```

File COMMON.FTN

```
      PARAMETER M=100
      COMMON X(M),Y(M)
```

5.1.3 OPEN and CLOSE Statements

The OPEN and CLOSE statements provide you with precise, explicit, and efficient control of I/O devices and files. Some examples follow:

```
OPEN (UNIT=1, STATUS='NEW', INITIALSIZE=200)
```

This statement creates a sequential file and allocates the space required for the file. Allocation of space at file opening is more efficient than dynamic extension of the file.

```
OPEN (UNIT=1, STATUS='UNKNOWN', EXTENDSIZE=200)
```

This statement specifies a relatively large EXTENDSIZE value, which is useful when a program writes many blocks to a file; it is faster to use one large extension than several small ones.

```
OPEN (UNIT=J, STATUS='NEW'...)
    .
    .
    .
IF (IERR) CLOSE(UNIT=J, STATUS='DELETE')
    .
    .
    .
CLOSE (UNIT=J, STATUS='SAVE')
```

If an error (denoted by IERR) occurs that makes the file created by the OPEN statement invalid or useless, the file is efficiently deleted.

```
     CHARACTER*40 FILNAM
1    TYPE 100
100  FORMAT('$INPUT  FILE?')
     ACCEPT 101,FILNAM
101  FORMAT (A)
     OPEN (UNIT=3, FILE=FILNAM, STATUS='OLD', ERR=9)
       .
       .
       .
9    TYPE 102, FILNAM
102  FORMAT (' ERROR OPENING FILE ',A)
     GO TO 1
```

This program fragment reads a file specification into the character variable FILNAM. The specified file is then opened for processing.

```
OPEN(UNIT=1,STATUS='NEW', ORGANIZATION='INDEXED',
     RECL=60,FORM='UNFORMATTED',
     KEY= (1:20, 30:33:INTEGER, 46:57), ACCESS='KEYED')
```

This statement creates a new indexed file that has three keys: the primary key is from byte 1 to byte 20; the first alternate key is an integer key from byte 30 to byte 33; and the second alternate key is from byte 46 to byte 57.

NOTE

> If you are adding several records to a file, make certain you specify a large enough EXTENDSIZE to reflect the size the file will be at the end of the program.

5.1.4 INTEGER*2 and INTEGER*4

Because the PDP–11 is a 16-bit computer, the code sequences generated for INTEGER*4 computations are larger and slower than those for their INTEGER*2 counterparts. Therefore, the use of INTEGER*4 should be limited to those data items requiring 32-bit representation; INTEGER*2 should be used elsewhere. In general, it is advisable to minimize use of the /I4 compiler option.

5.2 Compiler Optimizations

Optimization is producing the greatest amount of processing with the least amount of time and memory.

The primary goal of FORTRAN-77 optimization is to produce an object program that executes faster than an unoptimized version of the same source program. A secondary goal is to reduce the size of the object program.

The language elements you use in a source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization. The FORTRAN-77 compiler performs the following optimizations:

- Constant folding: Integer constant expressions are evaluated at compile-time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Argument-list merging: if two function or subroutine references have the same arguments, a single copy of the argument list is generated.
- Branch instruction optimizations for arithmetic and logical IF statements.
- Eliminating unreachable ("dead") code: an optional warning message is issued to indicate unreachable statements in a source program.
- Recognizing and replacing common subexpressions.
- Removing invariant computations from DO loops.
- Local register assignment: frequently referenced variables are retained (if possible) in registers to reduce the number of load and store instructions required.
- Assigning frequently used variables and expressions to registers across DO loops.
- Constant pooling: storage is allocated for only one copy of a constant in the compiled program. Constants, including most numeric constants, used as immediate-mode operands are not allocated storage.
- Inline code expansion for some intrinsic functions.
- Fast calling sequences for the real and double-precision versions of some intrinsic functions.

- Reordering the evaluation of expressions to minimize the number of temporary values required.
- Delaying unary minus and .NOT. operations to eliminate unary negation and complement operations.
- Partially evaluating Boolean expressions. For example, if e1 in the following expression has the value .FALSE., e2 is not evaluated:

  ```
  IF (e1.AND.e2) GO TO 20
  ```

  The order in which e1 and e2 appear in the source statement has no effect on partial evaluation.
- Peephole optimization of instruction sequences: examining code on an instruction-by-instruction basis to find operations that can be replaced by shorter, faster operations.

5.2.1 Characteristics of Optimized Programs

An optimized FORTRAN-77 program is computationally equivalent to an unoptimized program; therefore, identical numerical results are obtained and equivalent (in meaning, not quantity) run-time diagnostic messages are produced. An optimized program, however, can produce fewer run-time diagnostic messages and the diagnostics can occur at different statements in the source program.

Example 5-1: Effects of Optimization on Error Reporting

```
Unoptimized Program          Optimized Program

   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
```

In Example 5-1, if Y has the value 0.0, the unoptimized program produces 12 zero-divide errors at run time; the optimized program, however, produces only one zero-divide error because the calculation that produces the error has been moved out of a loop. (Note that t is a temporary variable created by the compiler.)

Note that optimizations such as eliminating redundant calculations and moving invariant calculations out of loops can affect the use of the ERRTST system subroutine. For example, in the above program, a call to ERRTST from inside the loop does not detect a zero-divide error in the loop calculation because the compiler has moved the error-producing part of the calculation outside the loop.

5.2.2 Compile-Time Operations on Constants

The compiler performs the following computations on expressions involving constants (including PARAMETER constants):

- Negation of constants: constants preceded by unary minus signs are negated at compile time. For example:

  ```
  X = -10.0
  ```

  is compiled as a single move operation.

- Type conversion of constants: Lower-ranked constants are converted to the data type of the higher-ranked operand at compile time. For example:

  ```
  X = 10*Y
  ```

  is compiled as:

  ```
  X = 10.0*Y
  ```

- Integer arithmetic on constants: Expressions involving +, -, *, / or ** operators are evaluated at compile time. For example:

  ```
  PARAMETER (NN=27)
  I = 2*NN+J
  ```

  is compiled as:

  ```
  I = 54+J
  ```

Array subscript calculations involving constants are simplified at compile time where possible. For example:

```
DIMENSION I(10,10)
I(1,2) = I(4,5)
```

is compiled as a single move instruction.

5.2.3 Source Program Blocks

FORTRAN-77 performs some optimizations only within the confines of a single "block" of a source program. A block is a sequence of one or more source statements. The start of a new block is generally defined by a labeled statement that is the target of a control transfer from another statement (for example, a GO TO, an arithmetic IF, or an ERR= option). An ENTRY statement also defines a new block. Some occurrences of statement labels do not define the start of a new block; these occurrences are as follows:

- Unreferenced statement labels.
- A label terminating a DO loop, provided the only references to the label occur in DO statements.
- Labels of FORMAT statements. FORMAT statements must be labeled, but control cannot be transferred to a FORMAT statement.
- Labels such that the only reference to the label occurs in the immediately preceding arithmetic IF statement. For example:

```
   IF(A) 10,20,20
10 X = 1.
```

- Singly referenced labels. A jump to a singly referenced label may be equivalent to an IF THEN/ENDIF structure. If it is, the IF THEN /ENDIF structure is used and the block is extended past the labeled statement.

The compiler imposes a limitation on the size of a single block. Therefore, a very long straight-line sequence of FORTRAN statements can be treated as several "blocks" during optimization.

A block can contain one or more DO loops, provided none of the labels within the loops defines the start of a new block. Therefore, the following are considered single blocks and are optimized as complete units:

```
      Example 1               Example 2

    X = B*C                   DO 20, I=1,N
    DO 10, I=1,N              DO 20, J=1,N
10  A(I) = A(I)/(B*C)         SUM = 0.0
    DO 20, J=1,N              DO 10, K=1,N
20  Y(J) = Y(J)+B*C       10  SUM = SUM+A(I,K)*B(K,J)
                          20  C(I,J) = SUM
```

If the label specified as the target of a GOTO in a logical IF is referenced only once, the structure may be equivalent to a block IF. For example, the following examples are equivalent:

```
              Example 1                Example 2

          IF (I .LT. J) GOTO 20    IF (I .LT. J)THEN
          A(I) = A(I)*J                A(I) = A(I)*J
          J=J-1                        J=J+1
      20  I=I+1                    ENDIF
                                   I=I+1
```

Even though these two examples are equivalent, Example 2 is more easily optimized. Therefore, as long as Example 1 is valid (that is, as long as both the GOTO and the label are in the same block, and the nesting rules are not violated), FORTRAN-77 transforms Example 1 into the form shown in Example 2.

Optimizations can be done most effectively over complete structures. If a block would otherwise be ended within either a block IF or DO structure, the block is instead ended at the beginning of the DO structure or the conditional block of the block IF structure.

A more thoroughly optimized object program is produced if the number of separate blocks is minimized. The common-subexpression, code motion, and register allocation optimizations are performed only within single blocks.

Multiple block IF structures, as well as nested DO and block IF structures, can occur within a single block.

5.2.4 Eliminating Common Subexpressions

Often a subexpression appears in more than one computation within a program. If the values of the operands of such a subexpression are not changed between computations, the value of the subexpression can be computed once and substituted for each occurrence of the subexpression. For example, B*C is a common subexpression in the following sequence:

```
A = B*C+E*F
    .
    .
    .
H = A+G-B*C
    .
    .
    .
```

```
     IF((B*C)-H)10,20,30
```

```
The preceding sequence is compiled as:

     t = B*C
     A = t+E*F
          .
          .
          .
     H = A+G-t
          .
          .
          .
     IF((t)-H)10,20,30
          .
          .
          .
```

where t is a temporary variable created by the compiler. Two computations of the subexpression B*C are eliminated from the sequence.

In the above example, you can modify the source program to eliminate the redundant calculation of (B*C). In the following example, however, you cannot reasonably modify the source program to achieve the same optimization ultimately effected by the compiler. The statements

```
     DIMENSION A(25,25), B(25,25)
     A(I,J)= B(I,J)
```

are compiled, without optimization, to a sequence of instructions of the form:

```
     t1 = J*25+I
     t2 = J*25+I
     A(t1) = B(t2)
```

where the variables t1 and t2 represent equivalent expressions. Recognizing the redundancy, the compiler optimizes the sequence into the following shorter, faster sequence:

```
     t = J*25 + I
     A(t) = B(t)
```

If a common subexpression is created within a conditional block of a block IF, this subexpression can be used anywhere within the conditional block in which it was created, including within any nested inner blocks; but it cannot be used outside that conditional block.

5.2.5 Removing Invariant Computations from Loops

Execution speed is enhanced if invariant computations are moved out of loops. For example, in the sequence

```
      DO 10, I=1,100
10    F = 2.0*Q*A(I)+F
```

the value of the subexpression 2.0*Q is the same during each iteration of the loop. Transformation of the sequence to:

```
      t = 2.0*Q
      DO 10, I=1,100
10    F = t*A(I)+F
```

moves the calculation 2.0*Q outside the body of the loop and eliminates 99 multiply operations.

However, invariant computations cannot be moved out of a zero-trip DO loop. For example, in the sequence

```
      DO 10, I=1,N
10    F=2.0*Q*A(I)+F
```

statement 10 is not executed for certain values of n; therefore, the invariant computation 2.0*Q cannot be moved out of the loop.

5.3 Run-Time Programming Considerations

You can often reduce the execution time of programs by making use of the following facts relevant to the FORTRAN–77 run-time environment.

- Unformatted I/O is substantially faster and more accurate than formatted I/O. The unformatted data representation usually occupies less file storage space as well. Therefore, you should use unformatted I/O for storing intermediate results on secondary storage.
- Specifying an array name in an I/O list is more efficient than using an equivalent implied DO list. A single I/O transmission call passes an entire array; however, an implied DO list can pass only a single array element for each I/O call.
- Implementing the BACKSPACE statement involves repositioning the file and scanning previously processed records. If a reread capability is required, it is more efficient to read the record into a temporary array and DECODE the array several times than to read and backspace the record.

- Array subscript checking is time-consuming and requires additional compiled code. It is primarily useful during program development and debugging.
- To obtain minimum direct access I/O processing, the record length should be an integer factor or multiple of the device block size of 512 bytes (for example, 32 bytes, 1024 bytes, and so on). Note that relative files under RMS-11 have additional overhead bytes added to each record.
- If the approximate size of the file is known, it is more efficient to allocate disk space when the file is opened than to incrementally extend the file as records are written.
- Using run-time formats should be minimized. The compiler preprocesses FORMAT statements into an efficient internal form. Run-time formats must be converted into this internal form at run-time. In many cases, variable format expressions allow the format to vary at run time as needed.
- RMS-11 I/O operations are substantially slower in most cases than corresponding FCS-11 I/O operations; therefore, using RMS-11 should generally be restricted to indexed files under keyed access.

5.4 FORTRAN-77 Optional Capabilities

The FORTRAN-77 system, as distributed, contains several optional capabilities supported by alternate OTS modules. These capabilities include:

- Running FORTRAN-77 without a Floating Point Processor
- Running FORTRAN-77 compiled programs under RSX-11S
- Choosing alternate run-time error reporting
- Obtaining an alternate floating point output conversion routine
- Building an OTS shareable library
- Building tasks with overlaid OTS modules
- Choosing an alternate random-number generator for compatibility with previous versions of the OTS (see Appendix B).

These options are described below. You should consult your system manager to determine the availability of these options; optional OTS modules are located in LB:[1,1] (LB: on RSTS/E). None of these options are required for normal use of the FORTRAN-77 system.

5.4.1 Non-FPP Operation (F77EIS.OBJ)

The FORTRAN-77 compiler does not require a floating point processor (FP11 or KEF11A) to compile a FORTRAN-77 program; the compiler can run on any PDP-11 with the EIS instruction set. However, the code generated by the FORTRAN-77 compiler is intended to run on a PDP-11 with FPP and may therefore contain FPP instructions.

A FORTRAN-77 source program containing no real, double-precision, or complex constants, variables, arrays, or function references is compiled into a PDP-11 program that contains no FPP instructions. If this program is linked using the module F77EIS.OBJ and the standard FORTRAN-77 OTS, as shown below, the resulting task executes no FPP instructions. Such programs can therefore run on any PDP-11 with the EIS instruction set.

```
    TKB INT/-FP=INT,LB:[1,1]F77EIS,LB:[1,1]F77FCS/LB
```

On RSTS/E, [1,1] is not included in the above command line.

If a compiled program unit contains no FPP instructions, the program listing contains the statement: NO FPP INSTRUCTIONS GENERATED.

5.4.2 RSX-11S Support (F7711S.OBJ)

An optional OTS module provides a subset of FORTRAN-77 I/O capability consistent with the facilities available in RSX-11S. Sequential I/O statements are supported for unit record devices such as terminals, non-spooled card readers, and line printers. This I/O support uses direct QIO operations and does not require any modules of the standard file system. The RSX-11S subset OTS is approximately 2000 words smaller than the normal OTS and can be provided as an object module or as a separate OTS library.

5.4.3 Optional OTS Error Reporting (F77NER.OBJ)

An optional OTS module that does not perform any run-time diagnostic message reporting is available; it is several hundred words smaller than the standard error-reporting module. Error processing and calls to ERRSET, ERRSNS, and ERRTST continue to operate normally, only the logging of the diagnostic message to the user terminal being suppressed. If this option is used, STOP and PAUSE messages are not produced.

5.4.4 Short Error Text (SHORT.OBJ)

For RSX–11M, RSX–11M–PLUS, and RSTS/E, the error message text for run-time error reports is contained in memory and requires over 1000 words. An alternative version is available that requires only one word. If the alternative is used, the error report is complete except for the 1-line English text description of the error. This module, $SHORT, is included in the task at task-build time. For example:

```
TKB> MAIN/FP=MAIN,LB:[1,1]F77FCS/LB:$SHORT,LB:[1,1]F77FCS/LB
```

On RSTS/E, [1,1] is not included in the above command line.

5.4.5 Intrinsic Function Name Mapping (F77MAP.OLB)

As discussed in Section 4.1, references to FORTRAN intrinsic functions are transformed at compile time into calls that use internal names. Therefore, if a program written in MACRO–11 uses a FORTRAN name instead of an internal name to reference an intrinsic function, an unresolved reference results during task build.

To prevent such unresolved references during the task building of a MACRO program, a set of concatenated object modules is provided for transforming FORTRAN–77 intrinsic-function names into internal names at task-build time. For example, the name SIN is transformed at task-build time by means of the following module:

```
      .TITLE    $MSIN
SIN   ::        JMP $SIN
      .END
```

The object module similar to the one for SIN is available for each intrinsic-function name.

An F77MAP library may be necessary to provide function mapping.

5.4.6 Floating Point Output Conversion (F77CVF.OBJ)

An alternative module for performing formatted output of floating point values under control of the D, E, F, and G format codes is provided. The standard module uses multiple-precision, fixed-point integer techniques to maintain maximum accuracy during the conversion. (FPP hardware is not used.) The alternative module performs the same functions using the FPP hardware; it is substantially faster but in some cases less accurate than the standard module. The standard module is accurate to 16 decimal digits; the optional module is accurate to 15 digits.

5.4.7 OTS Resident Libraries

F7FCLS.MAC, F7FRES.MAC, and F7SRES.MAC are MACRO–11 source files that contain global references to all OTS modules. You can use these files as a starting point in building an OTS resident library. Documentation in the files describes the OTS modules and such logical groups of modules as sequential I/O support and complex arithmetic. If your operating system supports memory management directives, these resident libraries provide a more extensive capability without sacrificing address space.

The procedures for building an OTS resident library are described by documentation in the file, in Chapter 13 of this manual, and in the *PDP–11 FORTRAN–77 Object Time System Reference Manual.*

NOTE

If the OTS resident library is overlaid, you must place all OTS I/O modules in the same overlay.

5.4.8 OTS Overlay Files

There are two OTS overlay files:

- FCS11M.ODL (FCS-11 support for RSX–11M/M–PLUS, RSTS/E, and VMS)
- RMS11M.ODL (RMS–11(K) support for RSX–11M/M–PLUS, RSTS/E, and VMS)

Each file is an ODL fragment file that you can use for overlaying the FORTRAN-77 OTS modules. Also, each file contains documentation that describes OTS options and procedures for using the file. The following example of an ODL file includes the FCS-11 overlaid OTS file in the overlay file described in Section 1.4 (on RSTS/E, [1,1] is not included):

```
          .ROOT     MAIN-OTSROT-*(A,B,C), OTSALL
A:        .FCTR     PRE
B:        .FCTR     PROC
C:        .FCTR     POST
@LB:[1,1]FCS11M
          .END
```

The factor "OTSROT" must be added to the root segment; the factor "OTSALL" must also be added as a co-tree. These factors are defined in the OTS overlay files listed above.

The following example of an ODL file includes the overlaid RMS-11 OTS file of the overlay file described in Section 1.4, as well as the RMS overlay file RMS11X (on RSTS/E, [1,1] is not included):

```
          .ROOT     MAIN-OTSROT-RMSROT-OVL,OTSALL,RMSALL
OVL:      .FCTR     *(PRE,PROC,POST)
@LB:[1,1]RMS11M
@LB:[1,1]RMS11X
          .END
```

The factors "OTSROT" and "RMSROT" must be added to the root segment; the factors "OTSALL" and "RMSALL" must also be added as co-trees.

See Section 1.4 for more information about overlaid programs.

5.5 RMS-11 Link and Run-Time Considerations

When RMS-11 is used with programs that are not overlaid, even modest-sized programs produce tasks that overflow the address space of the PDP-11. There are two possible solutions to this problem: Expand the task size such that it is large enough to accommodate the task, or make the program smaller by overlaying.

If the task is near or beyond the task size limit, the task build fails with a message indicating an oversize task.

Even if your program successfully links, you may encounter buffer-space problems at run time, indicated by FORTRAN-77 error message #41: "NO BUFFER ROOM."

If this message is encountered, try rerunning your program with a larger task increment, using (except on RSTS/E):

```
RUN/INC: value taskname
```

value
The amount of additional memory to be used for buffers.

The RUN command may fail if the /INC value makes the total task size too large. If the RUN command does fail, the only choices you have to get a successful run are to reduce the size of your program or to overlay your program.)

5.6 FCS Link and Run-Time Considerations

Under certain circumstances, the open-file buffers kept by FCS in PSECT $$FSR1 may become fragmented, causing the FORTRAN–77 OTS to produce, unexpectedly, the error message: "No Buffer Room."

One of the circumstances under which one of the open-file buffers can become fragmented is as follows: suppose a program specifies ACTFIL=2, to indicate that the program has at most two files open at any one time; FCS then allocates 1024 bytes for two 512-byte buffers in PSECT $$FSR1 (512 bytes is the largest possible device buffer size).

Suppose further that a logical unit is opened to a terminal, causing FCS to allocate an 80-byte buffer (that device's buffer size) in PSECT $$FSR1. Then another logical unit is opened to a disk file, causing FCS to allocate the next 512 bytes in PSECT $$FSR1 as a buffer for the disk file. Finally, the logical unit connected to the terminal is closed, resulting in the release, by FCS, of the 80-byte buffer in PSECT $$FSR1.

Any attempt to open a second disk file (resulting in a 512-byte buffer) now fails because PSECT $$FSR1 does not have 512 contiguous bytes. It has 80 free bytes, then 512 bytes in use by the first disk file, then 432 (512 - 80) free bytes.

Some possible solutions to the above situation are to specify a block size of 512 when opening the terminal; to open the first disk file before opening the terminal (if possible); or to specify ACTFIL=3, to allocate a larger $$FSR1 buffer.

Chapter 6

Using Character Data

The character data type facilitates the manipulation of alphanumeric data. You can use character data in the form of character variables, arrays, constants, and substrings.

6.1 Character Substrings

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

```
ROBERT WILLIAM BOB JACKSON
```

and you want to extract the substring BOB, specify the following:

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

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

```
NAME(:18)
```

the resulting substring is:

```
ROBERT WILLIAM BOB
```

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

```
NAME(16:)
```

encompasses:

```
BOB JACKSON
```

6.2 Character Constants

Character constants are strings of characters enclosed in apostrophes. You can assign a character value to a character variable in much the same way you would assign a numeric value to a real or integer variable. For example, as a result of the statement

```
XYZ = 'ABC'
```

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. Thus, if you specify:

```
CHARACTER*2 XYZ
```

```
XYZ = 'ABC'
```

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

```
CHARACTER*6 XYZ
```

```
XYZ = 'ABC'
```

result in having:

```
ABC
```

stored in XYZ. If the previous contents of XYZ were CBSNBC, the result would still be ABC because the previous contents are overwritten.

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

```
CHARACTER*17 TITLE
PARAMETER (TITLE = 'THE METAMORPHOSIS')
```

you can use the symbolic name TITLE anywhere a character constant is allowed.

You can include an apostrophe as part of the constant by specifying two consecutive apostrophes. For example, the statements

```
CHARACTER*15 TITLE
PARAMETER (TITLE = 'FINNEGANS''S WAKE')
```

result in the character constant FINNEGAN'S WAKE.

The value assigned to a character parameter can only be a character constant.

6.3 Declaring Character Data

To declare variables or arrays as character type, you 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 and AGE is defined as a 2-byte variable.

Character strings and character arrays are not interchangeable. Character strings comprise one or more characters; character arrays comprise one or more character strings.

Both must be declared and referenced uniquely to avoid compiler and run-time errors. This section describes how to declare and reference character strings and arrays correctly.

6.3.1 Character String Declaration

A character string declarator has the form

```
  CHARACTER[*len[,]] v[*len][,v[*len]] . . .
```

len
The length specification, that is, the number of characters in a character variable. Len must be an unsigned, nonzero, integer constant.

A length len immediately following the word CHARACTER is the length specification for each variable in the character string statement not having its own length specification. A length specification immediately following a variable is the length specification for only that variable. If a length is not specified for a variable, its length is 1.

v
A variable name.

The following examples illustrate the length specification rules when applied to character string declaration.

CHARACTER*10	FNAM	A string FNAM of ten characters
CHARACTER*10	FNAM,MNAM	Two strings FNAM and MNAM of ten characters each
CHARACTER	FNAM*10	A string FNAM of ten characters
CHARACTER	FNAM*10,MNAM*10	Two strings FNAM and MNAM of ten characters each
CHARACTER*2	FNAM*10	A string FNAM of ten characters
CHARACTER*2	FNAM*10,MNAM	A string FNAM of ten characters and a string MNAM of two characters
CHARACTER	FNAM	A string FNAM of one character

6.3.2 Character Array Declaration

A character array declarator has the form

```
CHARACTER[*len[,]] v(elm)[*len] [,v(elm)[*len]]
```

len
The length specification, that is, the number of characters in a character array element. Len must be an unsigned, nonzero, integer constant.

A length len immediately following the word CHARACTER is the length specification for each array element in the character array statement not having its own length specification. A length specification immediately following an array element is the length specification for only that array element. If a length is not specified for an array element, its length is 1.

v
An array name.

elm
The array declarator, that is, the number of elements in the array.

The following examples illustrate the length specification rules when applied to character array declaration.

CHARACTER*10	FNAM(5)	A 5-element array FNAM, each element has 10 characters
CHARACTER*10	FNAM(5),MNAM(5)	Two 5-element arrays FNAM and MNAM, each element has 10 characters
CHARACTER	FNAM(5)*10	A 5-element array FNAM, each element has 10 characters
CHARACTER	FNAM(5)*10,MNAM(5)*10	Two 5-element arrays FNAM and MNAM, each element has 10 characters
CHARACTER*2	FNAM(5)*10	a
CHARACTER*2	FNAM(5)*10,MNAM(5)	A 5-element array FNAM, each element has 10 characters; A 5-element array MNAM, each element has 2 characters
CHARACTER	FNAM(5)	A 5-element array FNAM, each element has 1 character

6.3.3 Character String Reference

A character string reference has the form

```
v([e1]:[e2])
```

v
A character variable.

e1
A numeric expression that specifies the leftmost character position of the string. Character positions within a character variable are numbered from left to right, beginning at one. If e1 is omitted, FORTRAN–77 assumes that e1 equals one.

e2
A numeric expression that specifies the rightmost character position of the string. Character positions within a character variable are numbered from left to right, beginning at one. If e2 is omitted, FORTRAN–77 assumes that e2 equals the length specification.

The following examples illustrate the character-range rules when applied to character-string references.

FNAM(7:9)	Characters 7 through 9 of string FNAM
FNAM(3:3)	Character 3 of string FNAM
FNAM(:6)	Characters 1 (default) through 6 of string FNAM
FNAM(2:)	Characters 2 through the length specification (default) of string FNAM
FNAM	All characters of string FNAM

6.3.4 Character Array Reference

A character array reference has the form

```
a(s[,s]...)([e1]:[e2])
```

a
A character array name.

s
A subscript expression.

e1
A numeric expression that specifies the leftmost character position of the string. Character positions within an array element are numbered from left to right, beginning with one. If e1 is omitted, FORTRAN–77 assumes the e1 equals one.

e2
A numeric expression that specifies the rightmost character position of the string. Character positions within an array element are numbered from left to right, beginning with one. If e2 is omitted, FORTRAN–77 assumes that e2 equals the length specification.

The following examples illustrate the character-range rules when applied to referencing character arrays.

FNAM(1)(1:10)	Characters 1 through 10 of the first string of array FNAM
FNAM(2)(3:3)	Character 3 of the second string of array FNAM
FNAM(3)(:4)	Characters 1 (default) through 4 of the third string of array FNAM
FNAM(4)(6:)	Characters 6 through the length specification (default) of the fourth string of array FNAM
FNAM(5)	All characters of the fifth string of array FNAM

6.3.5 Error 21 and Program Corrections

Error 21, Missing operator or delimiter symbol, is generated at compile time if an illegal reference is made to a character string. The following program demonstrates this error.

```
PROGRAM CHAR
CHARACTER*5 ASTR,BSTR*2
ASTR(1:1) = '1'
BSTR = ASTR(1)   ! This line generates ERROR 21 at compile time
END
```

The error is generated at the indicated line because ASTR is illegally referenced as an array, not as a string. The compiler expected to see ASTR referenced with the form ASTR(1:1). That is, a ":1" was expected after the "1" thus the missing operator or delimiter symbol error. The corrected program follows.

```
PROGRAM CHAR
CHARACTER*5 ASTR,BSTR*2
ASTR(1:1) = '1'
BSTR = ASTR(1:1)
END
```

There is an alternate way to correct this program. Instead of changing the reference to ASTR, you might change the declaration of ASTR (making it an array, as the original incorrect program assumed it was). However, the assignment to ASTR must be modified to reference the first (and only) string of the array. The second version of the program follows.

```
PROGRAM CHAR
CHARACTER*5 ASTR(1),BSTR*2
ASTR(1)(1:1) = '1'
BSTR = ASTR(1)   ! Accept defaults for range of elements
END
```

6.4 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, but that each array element in TEAM contains a 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.

6.5 Character Data Examples

An example of character data usage is shown in Example 6–1. The example is a program that manipulates the letters of the alphabet. The results are shown in Example 6–2.

6.6 Character Library Functions

The PDP–11 FORTRAN–77 system provides the following character functions:

- ICHAR
- INDEX
- LEN
- LGE, LGT, LLE, LLT

The following sections describe these functions.

6.6.1 ICHAR Function

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

```
ICHAR(c)
```

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

Example 6–1: Character Data Usage

```
       CHARACTER C,ALPHA*26
       DATA ALPHA/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
       WRITE(6,90)
90     FORMAT(' CHARACTER EXAMPLE PROGRAM OUTPUT')

       DO 10 I = 1:26
              WRITE(6,*) ALPHA
              C = ALPHA(1:1)
           ALPHA(1:25) = ALPHA(2:26)
           ALPHA(26:26) = C
10     CONTINUE

       CALL REVERS(ALPHA)
       WRITE(6,*) ALPHA

       CALL FIND('UVW',ALPHA)
       CALL FIND('AAA','DAAADHAJDAAAJAAA CEUEBCUEI')

       WRITE (6,*) ' END OF CHARACTER EXAMPLE PROGRAM'
       END

       SUBROUTINE REVERS(S)
       CHARACTER T*1,S*26

       K = 26
       DO 10 I = 1, K/2
              T = S(I:I)
              S(I:I) = S(K:K)
              S(K:K) = T
              K = K -1
10     CONTINUE
       RETURN
       END
```

Example 6–1 Cont'd. on next page

Example 6–1 (Cont.): Character Data Usage

```
        SUBROUTINE FIND(SUB,S)
        CHARACTER*3 SUB, S*26
        CHARACTER*132 MARKS

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

        WRITE(6,91) S, MARKS
91      FORMAT(2(/1X,A))
        RETURN
        END
```

6.6.2 INDEX Function

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

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

c1

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

c2

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

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

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

Example 6–2: Output Generated by Example Program

```
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

ZYXWVUTSRQPONMLKJIHGFEDCBA

DAAADHAJDAAAJAAA CEUEBCUEI
 #       #   #
 END OF CHARACTER EXAMPLE PROGRAM
```

If there are multiple occurrences of the substring, INDEX locates the first (left-most) one. Use of the INDEX function is illustrated in Examples 6–1 and 6–2. jacki

6.6.3 LEN Function

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

```
LEN(c)
```

c
A character expression.

6.6.4 LGE, LGT, LLE, and LLT Functions

The lexical comparison functions (LGE, LGT, LLE, and LLT) compare 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)
```

c1,c2
Character expressions.

You may wish to include these functions in FORTRAN programs that can be used on computers that do not use the ASCII character set. In PDP-11 FORTRAN-77, the lexical comparison functions are equivalent to the .GE., .GT., .LE., :LT. relational operators. For example, the statement

```
IF (LLE (string1, string2)) GO TO 100
```

is equivalent to:

```
IF (string1.LE.string2) GO TO 100
```

6.7 Character Input/Output

The character data type simplifies transmitting alphanumeric data. You can read and write character strings of any length from 1 to 255 characters. For example; the statements

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

cause 24 characters to be read from logical unit 12 and stored in the 24-byte character variable TITLE. If instead of character data you were to use Hollerith data stored in numeric variables or arrays, the following code is necessary:

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

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

Chapter 7

Using Indexed Files

This chapter provides detailed information on using indexed organization files. Included is an extended example. The indexed file is defined in Chapter 7 of the *PDP–11 FORTRAN–77 Language Reference Manual.*

Indexed organization is especially suitable for maintaining complex files from which records can be selected on the basis of one of several criteria. For example, a mail order firm using an indexed file to store its customer list might select records on the basis of a unique customer order number, the customer's zip code, or the item ordered. In such cases, reading sequentially on the basis of the zip code key produces a mailing list already sorted by zip code, and reading sequentially on the basis of the item-ordered key provides a list of customers sorted by the product ordered.

7.1 Accessing Indexed Files

You can access indexed files in both the sequential and the keyed modes. Sequential reading retrieves records in sorted order by defined key field. Keyed access, on the other hand, permits random record selection on the basis of a particular key-field value.

Once you select a record by key, a sequential read retrieves records with ascending key values, beginning with the key-field value of the initial indexed READ. Using the keyed and sequential access modes in combination is sometimes referred to as the Indexed Sequential Access Method (ISAM).

When you specify ACCESS="KEYED" in an OPEN statement, you enable both sequential and keyed access to an indexed file.

7.2 Creating an Indexed File

You can create an indexed file with the following:

- An OPEN statement
- An appropriate utility

You can use the OPEN statement to specify the more common file options and a utility to select features not directly supported from FORTRAN-77. Note, however, that any indexed file created with a utility can be accessed by FORTRAN-77 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, may also be defined; they are numbered from 1 through a maximum of 254. While an indexed file may have as many as 255 key fields defined, in practice few applications require more than three or four key fields.

When you design an indexed file, you decide which character positions within each record are to be the key fields. There are three key data types supported by PDP-11 FORTRAN-77: INTEGER*2, INTEGER*4, and CHARACTER. Using the example of a mail order firm, you might define a file record to consist of the following fields:

```
INTEGER*4 ORDER            ! Positions 1:4
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            ! Positions 66:74
INTEGER*2 ITEM             ! Positions 75:76
```

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,                ! 19 storage units
4  KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER),
5  ERR=9999)
```

This OPEN statement establishes the attributes of the file, including 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, respectively.

You may specify the KEY= keyword when opening an existing file; the FORTRAN Run-Time Library ensures that the given key specification matches that of the file.

FORTRAN uses RMS default key attributes when creating an indexed file. These defaults are as follows:

- Primary key values cannot be changed when a record is rewritten.
- Primary key values cannot be duplicated; that is, no two records can have the same primary key value.
- Alternate keys may both be changed and have duplicates.

You can use an RMS utility or a USEROPEN routine to override these defaults and to specify other values not supported by FORTRAN-77, such as null key values, key names, and key data types other than integer and character.

Refer to Section 2.3.12 for information on using the USEROPEN keyword in FORTRAN-77 OPEN statements. The *RMS-11 User's Guide* has more information on indexed file options.

7.3 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 value. Subsequent sequential read operations cause the next-record pointer to be the one with the next higher key value in the same key field. In case of duplicate key 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, that is, the record that is locked from access by other programs sharing the file. The current record can be operated on by the REWRITE statement and the DELETE statement, but 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.

In addition, an error results if a REWRITE or DELETE operation is performed when the current-record pointer is undefined.

7.4 Writing to Indexed Files

You can write records to an indexed file with either formatted or unformatted indexed WRITE statements. Each WRITE inserts a new record into the file and updates the index(es) so that the new record appears in the correct order for each key field.

Continuing the mail order file example of Section 7.2, you could add a new record to the file with the following statement:

```
WRITE (UNIT=10,ERR=9999) ORDER,
1 NAME,ADDRESS,CITY,STATE,ZIP,ITEM
```

7.4.1 Duplicate Keys

It is possible to write two or more records with the same key value. Whether this duplicate-key situation is allowed depends on the attributes that were specified for the file when it was created. By default, FORTRAN-77 creates files that allow duplicate alternate keys but that prohibit duplicate primary keys (see Section 7.2). If duplicate keys are present in a file, the records with equal keys are retrieved on a first-in, first-out basis.

For example, assume that five records are written to an indexed file in the following order (for clarity, only key fields are shown):

Order	Zip	Item
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), the sorted order of the records is unaffected by the duplicated ITEM key, as shown below:

Order	Zip	Item
903	14853	375
942	02163	2736
1023	70856	375
1263	33032	690
1348	44901	1047

If the file is read along the second alternate key (ITEM), however, the sort order is affected by the duplicate key, as shown below:

Order	Zip	Item
1023	70856	375
903	14853	375
1263	33032	690
1348	44901	1047
942	02163	2736

Notice that the records containing the same key value (375) were retrieved in the order in which they were written to the file.

7.4.2 Omitting Alternate Keys

You can omit one or more alternate keys when writing to an indexed file that contains variable-length records. To omit any alternate key field, omit the alternate key-field name from the WRITE statement. However, do not specify another field after that point; an omitted key must be at the end of the variable-length record. For example, if the last record in the mail order example (ORDER 1263) was written with the statement

```
WRITE (UNIT=10,ERR=9999) ORDER,
1 NAME,ADDRESS,CITY,STATE,ZIP
```

then the result of reading the complete file along the alternate ITEM index would be as follows:

Order	Zip	Item
1023	70856	375
903	14853	375
1348	44901	1047
942	02163	2736

Because the ITEM was omitted when the last record was written, there is no index entry for that key; and it cannot be read when the file is sorted on ITEM.

You may omit only alternate keys from a record; the primary key must always be present.

7.5 Reading From Indexed Files

You can read records in an indexed file with either sequential or indexed READ statements.

Indexed READ statements position the file pointers (see Section 7.3) at a particular record (determined by the key value), the key-of-reference, and the match criterion. Once you retrieve a particular record by key, you can use sequential READ statements to retrieve records with increasing key values.

The following FORTRAN-77 program segment prints the order number and zip code of each record, with a zip code in which the first 5 characters are greater than or equal to '10000' but less than '50000':

```
C
C   Read first record with ZIP key greater than or
C   equal to '10000'.
C
      READ (UNIT=10, KEYGE='10000', KEYID=1,  ERR=9999),
      1  ORDER, NAME, ADDRESS, CITY, STATE, ZIP
C
C   While the Zip Code previously read is within range, print
C   the order number and zip code, then read the next record.
C
10    IF (ZIP .LT. '50000') THEN
        PRINT *, 'Order number', ORDER, 'has zip code',
      1   ZIP
        READ (UNIT=10,  END=200, ERR=9999)
      1   ORDER, NAME, ADDRESS, CITY, STATE, ZIP
C
C   END= branch will be taken if there are no more records
C   in the file.
C
        ENDIF
        GOTO 10
200   CONTINUE
```

The error branch on the indexed READ in the 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. However, if the sequential READ has accessed all records in the file, an end-of-file status occurs, just as it does with other file organizations.

7.6 Updating Records

You use the REWRITE statement to update existing records in an indexed file. You cannot replace an existing record simply by writing it again: A WRITE statement attempts 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 a REWRITE statement. As an example, to update the record containing ORDER 903 (see prior examples) so that the NAME field becomes 'Theodore Zinck', you might use the following FORTRAN-77 code segment:

```
READ (UNIT=10, KEY=903, KEYID=0, IOSTAT=IOS, ERR=9999)
1  ORDER, NAME, ADDRESS, CITY, STATE, ZIP; ITEM
NAME='Theodore Zinck'
REWRITE (UNIT=10,  ERR=9999) ORDER
1  NAME, ADDRESS, CITY, STATE, ZIP, ITEM
```

When you rewrite a record, key fields may change. Whether a key-field change is permitted depends on the attributes given the file when it was created.

7.7 Deleting Records

To delete records from an indexed file, you use the DELETE statement. The DELETE and REWRITE statements are similar in that each operates on a record that has been locked by a READ statement.

The following FORTRAN-77 code segment deletes the second record in the file with ITEM 375 (refer to previous examples):

```
READ (UNIT=10, KEY=375, KEYID=2,  ERR=9999)
READ (UNIT=10,  ERR=9999) ORDER
1  NAME, ADDRESS, CITY, STATE, ZIP, ITEM
IF (ITEM .EQ. 375) THEN
   DELETE (UNIT=10,  ERR=9999)
ELSE
   PRINT *,'There is no second record.'
ENDIF
```

Deletion removes a record from all defined indexes in the file.

7.8 Using Integer Keys

When writing an integer-key value to a record (with an indexed WRITE statement), use an A2 format for an INTEGER*2 value and an A4 format for an INTEGER*4 value. Do not use an I format, because the I format produces an ASCII representation that an indexed READ statement cannot later read.

To read a key field, you may use any format you wish, because the format you associate with an indexed READ has no bearing on the matching process used to locate the record in which the desired key field is located.

The following program segment is an example using integer keys with an indexed file. Note that ACODE and TEL, which are the third and second alternate keys in the record described below, are of type INTEGER*2 and INTEGER*4, respectively, and that the formats used to write these keys are A2 and A4, respectively.

The record layout is as follows:

Field	Size	Type	Meaning
FI	1	CHAR	First Initial
NAME	10	CHAR	Last Name
STADDR	20	CHAR	Street Address
CITY	10	CHAR	City
STATE	2	CHAR	State
SSN	9	CHAR	Social Security Number
ACODE	2	INT*2	Area Code
TEL	4	INT*4	Telephone Number
AGE	2	INT*2	Age

The keys are as follows:

PRIMARY	SSN	44:52
ALTERNATE 1:	NAME	2:11
ALTERNATE 2:	TEL	55:58:INTEGER
ALTERNATE 3:	ACODE	53:54:INTEGER

```
      CHARACTER FI*1,NAME*10,STADDR*20,CITY*10,STATE*2,SSN*9
      INTEGER*4 TEL
      INTEGER*2 AGE, ACODE
      COMMON /DBREC1/ACODE,TEL,AGE
      COMMON /DBREC2/NAME,FI,STADDR,CITY,STATE,SSN
      INTEGER*4 INTKEY

      OPEN(UNIT=1,NAME='DB.DAT',ORGANIZATION='INDEXED',ACCESS='KEYED',
      1 RECORDTYPE='FIXED',RECL=128,FORM='FORMATTED',TYPE='NEW',
      2 KEY=(44:52, 2:11, 55:58:INTEGER, 53:54:INTEGER))
          .
          .
          .
      WRITE(1,1000)FI,NAME,STADDR,CITY,STATE,SSN,ACODE,TEL,AGE
C
```

```
C   READ WITH KEY EQUAL TO INTKEY
C
        READ(1,1000,KEY=INTKEY,KEYID=IKEYID)
        1       FI,NAME,STADDR,CITY,STATE,SSN,ACODE,TEL,AGE
                .
                .
                .
C   READ WITH KEY GREATER THAN INTKEY
C
        READ(1,1000,KEYGT=INTKEY,KEYID=IKEYID)
        1       FI,NAME,STADDR,CITY,STATE,SSN,ACODE,TEL,AGE
                .
                .
                .
C   READ WITH KEY EQUAL TO OR GREATER THAN INTKEY
C
        READ(1,1000,KEYGE=INTKEY,KEYID=IKEYID)
        1       FI,NAME,STADDR,CITY,STATE,SSN,ACODE,TEL,AGE
                .
                .
                .
1000    FORMAT (A1,10A1,20A1,10A1,2A1,9A1,A2,A4,A2)

        STOP
        END
```

7.9 Error Conditions

You may encounter certain error conditions when using indexed files. The two most common conditions result from attempts to read locked records and attempts to create duplicate primary 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 can 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, the program can go back to the READ statement. For example:

```
      PARAMETER (LOCKED=52)
100   READ (UNIT=10, ERR=200) DATA
          .
          .
          .
200   CALL ERRSNS(IERR)
      IF(IERR .EQ. LOCKED) GOTO 100
```

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 error condition, creation of duplicate primary keys, occurs when a program tries to create a record with a key value that is already in use. To handle this situation, you might have your program prompt for a new key value whenever an attempt is made to create a duplicate key. This technique is demonstrated below:

```
      INTEGER DUPKEY
      PARAMETER (DUPKEY=50)

200   WRITE(UNIT=10, ERR=300) KEYVAL, DATA
          .
          .
          .
300   CALL ERRSNS(IERR)
      IF (IERR .EQ. DUPKEY) THEN
          TYPE*, 'This key value already exist. Please enter'
          TYPE*, 'a different key value, or press CONTROL Z'
          TYPE*, 'to discontinue this operation.'
          READ(UNIT=*, END=999) KEYVAL
          GOTO 200
      ELSE
          TYPE*, 'ERROR',IERR,'DURING WRITE'
          STOP
      ENDIF
999   CONTINUE
```

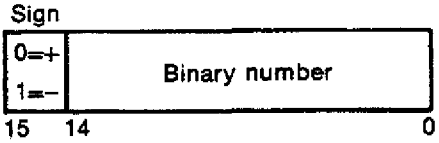
Appendix A

FORTRAN-77 Data Representation

A.1 Integer Formats

The following sections display the formats for INTEGER*2 and INTEGER*4.

A.1.1 INTEGER*2 Format

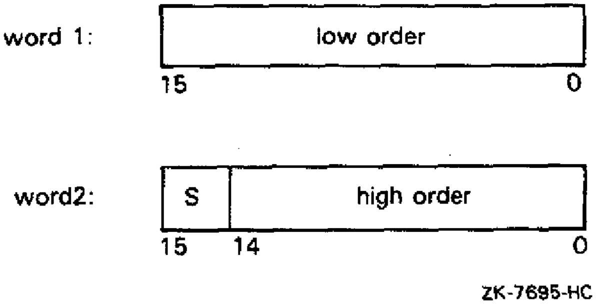


ZK-1244-83

Integers are stored in two's complement representation. INTEGER*2 values lie in the range -32768 to +32767. For example:

```
+22 = 000026   (octal)
 -7 = 177771   (octal)
```

A.1.2 INTEGER*4 Format



INTEGER*4 values are stored in two's complement representation. The first word contains the low-order part of the value; the second word contains the sign and high-order part of the value. If the value is in the range of an INTEGER*2 value (-32768 to +32767), then the first word may be referenced as an INTEGER*2 value.

A.2 Floating Point Formats

The exponent for both 2-word and 4-word floating point formats is stored in excess-128 notation. Binary exponents from -128 to +127 are represented by the binary equivalents of 0 through 255. Fractions are represented in sign-magnitude notation, with the binary radix point to the left. Numbers are assumed to be normalized; therefore, because it would be redundant, the most significant bit is not stored (the practice of not storing the most significant bit is called "hidden bit normalization"). The unstored bit is assumed to be a 1 unless the exponent is 0 (corresponding to 2**-128), in which case the unstored bit is assumed to be 0. The value 0 is represented by an exponent field of 0 and a sign bit of 0. For example, +1.0 would be represented in octal by:

```
40200
0
```

in the 2-word format, or:

```
40200
0
0
0
```

in the 4-word format. The decimal number -5.0 is:

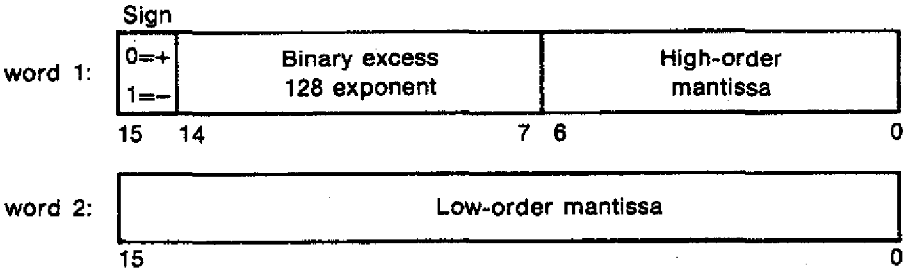
```
140640
0
```

in the 2-word format, or:

```
140640
0
0
0
```

in the 4-word format.

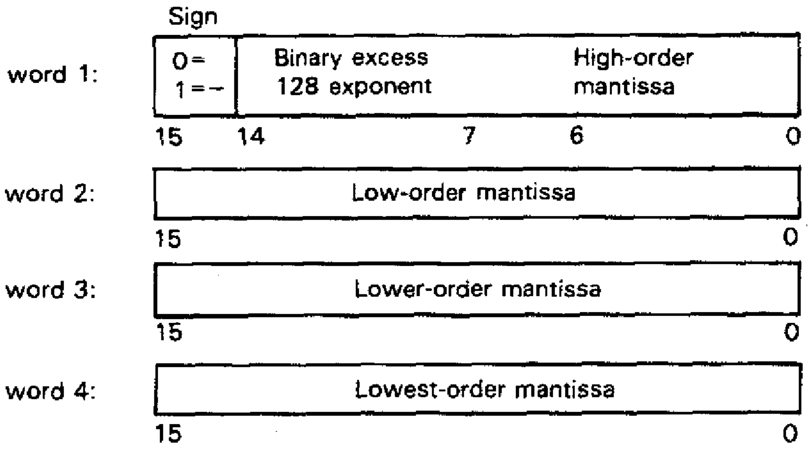
A.2.1 Real (REAL*4) Format (2-Word Floating Point)



ZK-1245-83

The form of a single-precision real number is sign magnitude, with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 15:0 in the second word a normalized 24-bit fraction with the redundant most significant fraction bit not represented. The value of a single-precision real number is in the approximate range .29*10**-38 through 1.7*10**38. The precision is approximately one part in 2**23–or typically seven decimal digits.

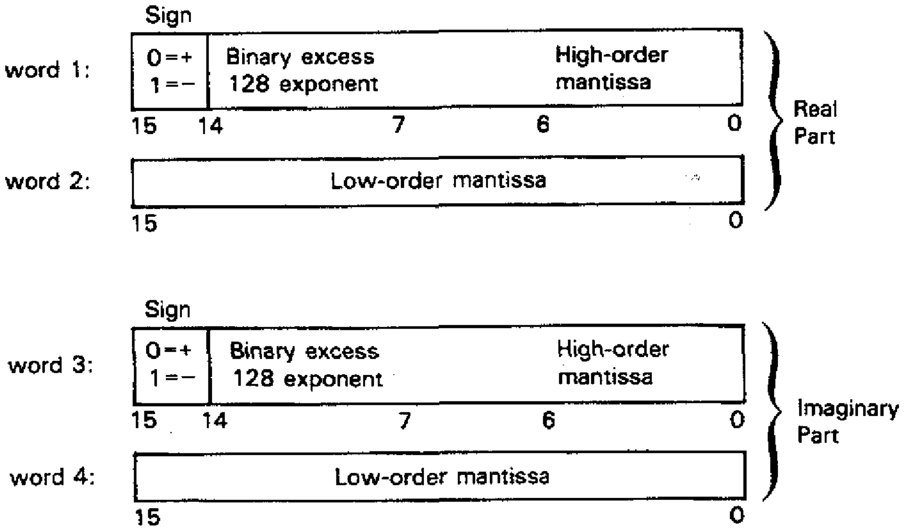
A.2.2 Double-Precision (REAL*8) Format (4-Word Floating Point)



ZK-7696-HC

The form of a double-precision real number is identical to that of a single-precision real number except for an additional 32 low-significance fraction bits. The exponent conventions and approximate range of values are the same as for a single-precision real value. The precision is approximately one part in 2**55–or typically 16 decimal digits.

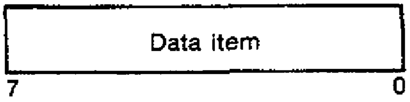
A.2.3 Complex Format



ZK-7698-HC

The form of a complex number is an ordered pair of real numbers. The first real number represents the real part of the imaginary number; the second represents the imaginary part.

A.3 LOGICAL*1 (BYTE) Format

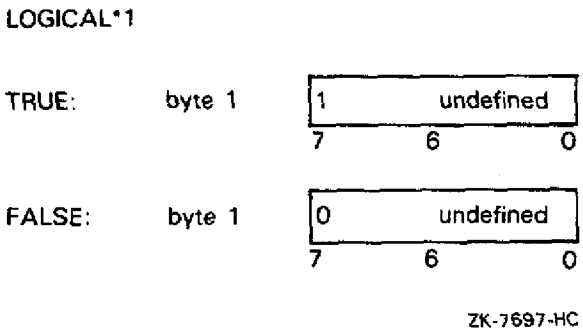


ZK-1246-83

The logical values true or false (see Section A.4), a single Hollerith character, or integers in the range of numbers from +127 to -128 can be represented in LOGICAL*1 format. LOGICAL*1 array elements are stored in adjacent bytes.

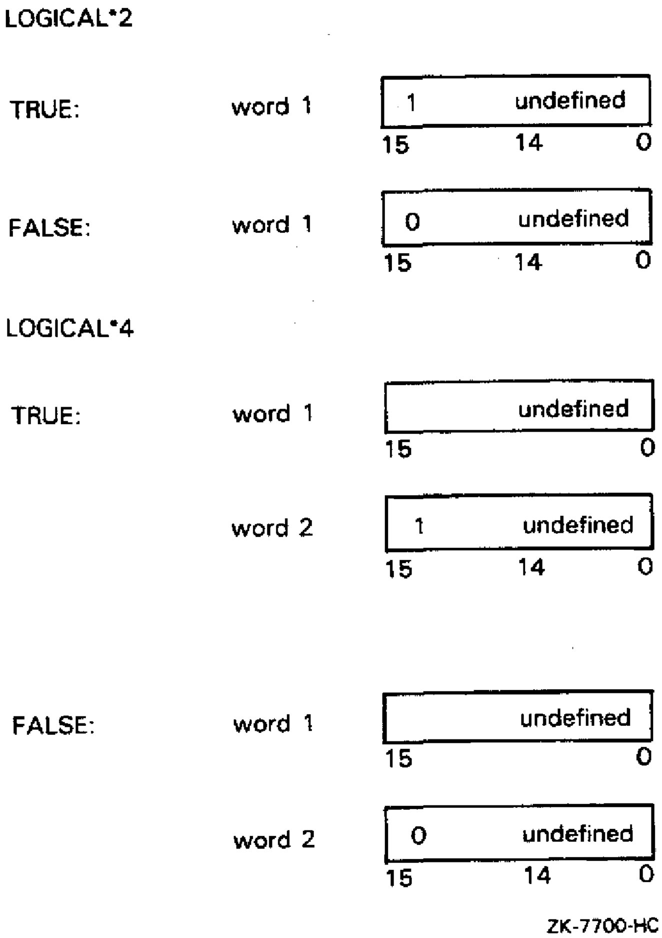
A.4 Logical Formats

LOGICAL*1



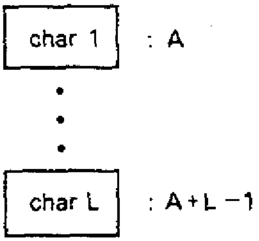
LOGICAL*2

LOGICAL*4



A.5 Character Representation

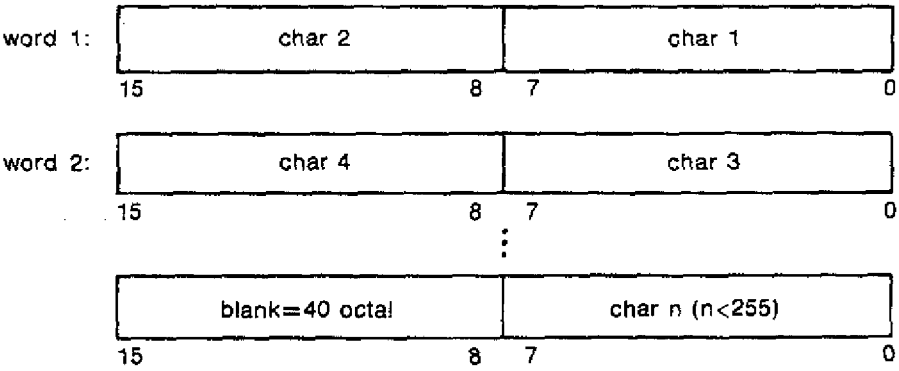
A character string is a contiguous sequence of bytes in memory.



ZK-7699-HC

A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 1 through 255.

A.6 Hollerith Format



ZK-1247-83

Hollerith constants are stored one character per byte. Hollerith values are padded on the right with blanks, if necessary, to fill the associated data item.

A.7 Radix–50 Format

Radix–50 character set

Value (Octal)	Octal ASCII	Character Equivalent Radix–50
(space)	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
(unused)		35
0-9	60-71	36-47

Radix–50 values are stored, up to three characters per word, by packing the Radix–50 values into single numeric values according to the formula:

```
((i*50+j)*50+k)
```

i,j,k
The code values of three Radix–50 characters.

The maximum Radix–50 value is, therefore:

```
47*50**2+47*50+47=174777(8)
```

The following table provides a convenient means of translating between the ASCII character set and Radix–50 equivalents. For example, given the ASCII string X2B, the Radix–50 equivalent is (arithmetic is performed in octal):

```
  X=113000
  2=002400
  B=000002
X2B=115402
```

Single Character or First Character	Second Character	Third Character
000000	000000	000000 (space)
A 003100	A 000050	A 000001
B 006200	B 000120	B 000002
C 011300	C 000170	C 000003
D 014400	D 000240	D 000004
E 017500	E 000310	E 000005
F 022600	F 000360	F 000006
G 025700	G 000430	G 000007
H 031000	H 000500	H 000010
I 034100	I 000550	I 000011
J 037200	J 000620	J 000012
K 042300	K 000670	K 000013
L 045400	L 000740	L 000014
M 050500	M 001010	M 000015
N 053600	N 001060	N 000016
O 056700	O 001130	O 000017
P 062000	P 001200	P 000020
Q 065100	Q 001250	Q 000021
R 070200	R 001320	R 000022
S 073300	S 001370	S 000023
T 076400	T 001440	T 000024
U 101500	U 001510	U 000025
V 104600	V 001560	V 000026
W 107700	W 001630	W 000027
X 113000	X 001700	X 000030
Y 116100	Y 001750	Y 000031

Single Character or First Character	Second Character	Third Character
Z 121200	Z 002020	Z 000032
$ 124300	$ 002070	$ 000033
. 127400	. 002140	. 000034
132500	002210	000035 (unused)
0 135600	0 002260	0 000036
1 140700	1 002330	1 000037
2 144000	2 002400	2 000040
3 147100	3 002450	3 000041
4 152200	4 002520	4 000042
5 155300	5 002570	5 000043
6 160400	6 002640	6 000044
7 163500	7 002710	7 000045
8 166600	8 002760	8 000046
9 171700	9 003030	9 000047

Appendix B

Algorithms for Approximation Procedures

This appendix contains brief descriptions of the algorithms used in intrinsic functions that involve approximations.

Some of the descriptions below give relative error bounds. These relative error bounds are for the approximating polynomials involved in the algorithms, and assume exact arithmetic. Possible additional sources of errors not reflected in these error bounds are:

- Rounding and truncation errors that can occur when a given argument is reduced to the range in which approximations for a polynomial or rational fraction are valid
- Rounding errors that can occur as a result of using finite-precision, floating point arithmetic in polynomial or rational-fraction computations

B.1 Real-Value Procedures

B.1.1 ACOS—Real Floating Point, Arc Cosine

ACOS(X) is computed as:

```
If X = 0, then ACOS(X) = pi/2
If X = 1, then ACOS(X) = 0
If X = -1, then ACOS(X) = pi
If 0 < X < 1, then ACOS(X) = ATAN(SQRT(1-X**2)/X)
If -1 < X <0, then ACOS(X) = ATAN(SQRT(1-X**2)/X) + pi
If 1 <  ABS(X) , error
```

B.1.2 DACOS—Double-Precision Floating Point Arc Cosine

DACOS(X) is computed as:

```
If X = 0, then DACOS(X) = pi/2
If X = 1, then DACOS(X) = 0
If X = -1, then DACOS(X) = pi
If 0 < X < 1, then DACOS(X) = DATAN(DSQRT(1-X**2)/X)
If -1 < X < 0, then DACOS(X) = DATAN(DSQRT(1-X**2)/X) + pi
If 1 <  ABS(X), error
```

B.1.3 ASIN—Real Floating Point Arc Sine

ASIN(X) is computed as:

```
If X = 0, then ASIN(X) = 0
If X = 1, then ASIN(X) = pi/2
If X = -1, then ASIN(X) = -pi/2
If 0 <  ABS(X) < 1, then ASIN(X) = ATAN(X/SQRT(1-X**2))
If 1 <  ABS(X), error
```

B.1.4 DASIN—Double-Precision Floating Point Arc Sine

DASIN(X) is computed as:

```
If X = 0, then DASIN(X) = 0
If X = 1, then DASIN(X) = pi/2
If X = -1, then DASIN(X) = -pi/2
If 0 <  ABS(X) < 1, then DASIN(X) = DATAN(X/DSQRT(1-X**2))
If 1 <  ABS(X), error
```

B.1.5 ATAN—Real Floating Point Arc Tangent

ATAN(X) is computed as:

1. If X < 0, then:

```
Begin
     Perform Steps 2, 3, and 4 with arg =  ABS(X)
     Negate the result since ATAN(X) = -ATAN(-X)
     Return End
```

2. If ABS(X) > 1, then:

```
Begin
     Perform Steps 3 and 4 with arg = 1/ABS(X)
     Negate result and add a bias of pi/2 since
     ATAN(ABS(X)) = pi/2 - ATAN(1/ABS(X))
     Return End
```

3. At this point the argument is 1 > = X > = 0
   If ABS(X) > TAN(pi/12), then:

```
Begin
     Perform Step 4 with arg = (X * SQRT(3) - 1)/
             (SQRT(3) + X)
     Add pi/6 to the result
     Return End
```

   Note: (X * SQRT(3) -1)/(X + SQRT(3)) <= TAN(pi/12) for ABS(X) > = TAN(pi/12)

4. Finally, the argument is ABS(X) <= TAN(pi/12)

```
Begin
     ATAN(X) = X * SUM(C[i] * X**(2[i])), i = 0:4
     Return End
```

The coefficients C[i] are drawn from Hart #4941.[1]
The relative error is <= 10**-9.54.

[1] 1. Hart, J. F. et al., *Computer Approximations* (John Wiley & Sons, 1968), P. 267

B.1.6 ATAN2—Real Floating Point Arc Tangent with Two Parameters

ATAN2(X,Y) is computed as:

```
If Y = 0 or X/Y > 2**25, ATAN(X,Y) = pi/2 * (sign X)
If Y > 0 and X/Y <= 2**25, ATAN2(X,Y) = ATAN(X/Y)
If Y < 0 and X/Y <= 2**25, ATAN2(X,Y) = pi * (sign X)
        + ATAN(X/Y)
```

B.1.7 DATAN—Double-Precision Floating Point Arc Tangent

DATAN(x) is computed as:

1. If X < 0, then:

```
Begin
     Perform Steps 2, 3, and 4 with arg =  ABS(X)
     Negate the result since DATAN(X) = -DATAN(-X)
     Return
End
```

2. If ABS(X) > 1, then:

```
Begin
     Perform Steps 3 and 4 with arg = 1/ABS(X)
     Negate result and add a bias of pi/2 since
     DATAN(ABS(X)) = pi/2 - DATAN(1/ABS(X))
     Return
End
```

3. At this point the argument is 1 > = X > = 0
   If ABS(X) > DATAN(pi/12) then:

```
Begin
     Perform Step 4 with arg = (X*DSQRT(3) - 1)/
     (DSQRT(3) + X)
     Add pi/6 to the result
     Return
End
```

   Note: (X*DQRT(3) -1)/(X + DQRT(3)) <= DATAN(pi/12) for AB(X) > = DATAN(pi/12)

4. Finally, the argument is ABS(X) <= DATAN(pi/12):

```
Begin
     DATAN(X) = X * SUM(C[i] * X**(2*i)), i = 0:8
     Return
End
```

The coefficient C[i]'s are drawn from Hart #4941.[1]
The relative error is <= 10**-9.54.

B.1.8 DATAN2—Double-Precision Floating Point Arc Tangent with Two Parameters

```
If Y = 0 or X/Y > 2**25, DATAN2(X,Y) = pi/2 * (sign X)
If Y > 0 and X/Y <= 2**25, DATAN2(X,Y) = DATAN(X/Y)
If Y < 0 and X/Y <= 2**25, DATAN2(X,Y) = pi * (sign X)
        + DATAN(X/Y)
```

B.1.9 ALOG10—Real Floating Point Common Logarithm

ALOG10(x) is computed as:

```
ALOG10(E) * ALOG(X)
```

where:

E = 2.718, the base of the natural log system.

See the description of ALOG (Section B.1.21) for the complete algorithm.

B.1.10 DLOG10—Double-Precision Floating Point Common Logarithm

DLOG10(X) is computed as:

```
DLOG10(E) * DLOG (X)
```

where:

E = 2.718, the base of the natural log system.

See the description of DLOG (Section B.1.22) for the complete algorithm.

[1] Hart, *Computer Approximations* p. 267.

B.1.11 COS—Real Floating Point Cosine

COS(X) is computed as:

```
SIN(X+pi/2)
```

See the description of SIN (Section B.1.23) for the complete algorithm.

B.1.12 DCOS—Double-Precision Floating Point Cosine

DCOS(X) is computed as:

```
DSIN(X+pi/2).
```

See the description of DSIN (Section B.1.24) for the complete algorithm.

B.1.13 EXP—Real Floating Point Exponential

EXP(X) is computed as:

```
If X > 88.028, overflow occurs
If X <= -88.5, EXP(X) = 0
If ABS(X) < 2**-28, EXP(X) = 1
```

Otherwise:

```
EXP(X) = 2**Y * 2**Z * 2**W
```

where:

```
Y = INTEGER(X*LOG2(E))
V = FRAC(X*LOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16
```

```
         P+wQ
2**W =
         P-wQ
```

P and Q are first degree polynomials in W**2. The coefficients of P and Q are drawn from Hart #1121.[1]

Powers of 2**(1/16) are obtained from a table. All arithmetic is done in double precision and then rounded to single precision at the end of calculation. The relative error is less than or equal to 10**-16.4.

[1] Hart, *Computer Approximations*, p. 206.

B.1.14 DEXP—Double-Precision Floating Point Exponential

See the description of EXP (Section B.1.13). The approximation is identical except that there is no conversion to single precision at the end.

B.1.15 COSH—Real Floating Point Hyperbolic Cosine

COSH(X) is computed as:

```
If  ABS(X)  < 2**-11, COSH(X) = 1

If 2**-11 <= ABS(X)  < 0.25,
     COSH(X) = DIGITAL's approximation[1]
If 0.25 <= ABS(X) <= 87.0,
     COSH(X) = (EXP(X) + EXP(-X))/2
If 87.0 <  ABS(X)  and  ABS(X)  - LOG(2) < 87,
     COSH(X) = EXP(ABS(X)  - LOG(2))

If 87.0 <  ABS(X)  and  ABS(X)  - LOG(2)>= 87, then overflow
```

B.1.16 DCOSH—Double Floating Point Hyperbolic Cosine

DCOSH(X) is computed as:

```
If  ABS(X)  < 2**-27, DCOSH(X) = 1

If 2**-27 <= ABS(X) < 0.25,
     DCOSH(X) = DIGITAL's approximation[1]

If 0.25 <= ABS(X) <= 87.0,
     DCOSH(X) = (DEXP(X) + DEXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     DCOSH(X) = DEXP(ABS(X) - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2)>= 87, then overflow
```

[1] This approximation is proprietary.

B.1.17 SINH—Real Floating Point Hyperbolic Sine

SINH(X) is computed as:

```
If  ABS(X)  < 2**-11, SINH(X) = X

If 2**-11 <= ABS(X) < 0.25,
     SINH(X) = DIGITAL's approximation[1]

If 0.25 <= ABS(X) <= 87.0,
     SINH(X) = (EXP(X) - EXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     SINH(X) = sign(X) * EXP(ABS(X)  - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2)>= 87, then overflow
```

B.1.18 DSINH—Double-Precision Floating Point Hyperbolic Sine

DSINH(x) is computed as:

```
If ABS(X) < 2**-27, DSINH(X) = X

If 2**-27 <= ABS(X) < 0.25,
     DSINH(X) = DIGITAL's approximation[1]

If 0.25 <= ABS(X) <= 87.0,
     DSINH(X) = (DEXP(X) - DEXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     DSINH(X) = sign(X) * DEXP(ABS(X) - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2)>= 87, then overflow
```

B.1.19 TANH—Real Floating Point Hyperbolic Tangent

TANH(X) is computed as:

```
If ABS(X) <= 2**-14, then TANH(X) = X

If 2**-14 < ABS(X) <= 0.25, then TANH(X) = SINH(X) / COSH(X)

If 0.25 < ABS(X) < 16.0, then
     TANH(X) = (EXP(2*X) - 1)/(EXP(2*X) + 1)

If 16.0 <= ABS(X), then TANH(X) = sign(X) * 1
```

[1] This approximation is proprietary.

B.1.20 DTANH—Double-Precision Floating Point Hyperbolic Tangent

DTANH(X) is computed as:

```
If ABS(X) <= 2**-14, then DTANH(X) = X

If 2**-14 < ABS(X) <= 0.25, then DTANH(X) = DSINH(X)/DCOSH(X)

If 0.25 < ABS(X) < 16.0, then
    DTANH(X) = (DEXP(2*X) - 1)/(DEXP(2*X) + 1)

If 16.0 <= ABS(X), then DTANH(X) = sign(X) * 1
```

B.1.21 ALOG—Real Floating Point Natural Logarithm

ALOG(x) is computed as:

```
If X <= 0, an error is signaled.

Therefore, let X = Y * (2**A)
```

where:

```
   1/2 <= Y < 1

Then LOG(X) = A * LOG(2) + LOG(Y)

If ABS(X-1) <= 0.25, let W = (X-1)/(X+1)

Then, LOG(X) = W * SUM(C[i] * W**(2*i))

Otherwise, let W = (Y-SQRT(2)/2)/(Y+SQRT(2)/2)

Then, LOG(X) = A * LOG(2) - 1/2 * LOG(2) +
     W * SUM C[i] * W**(2*i)
```

The coefficients are drawn from Hart #2662.[1]

The polynomial approximation used is of degree 4.

The relative error is less than or equal to 10**-9.9.

[1] Hart, *Computer Approximations*, p. 227.

B.1.22 DLOG—Double-Precision Floating Point Natural Logarithm

DLOG(x) is computed as:

```
If X <= 0, an error is signaled.

Therefore, let X = Y * (2**A)
```

where:

```
   1/2 <= Y < 1

Then, DLOG(X) = A * DLOG(2) + DLOG(Y)

If  ABS(X-1)  <= 0.25, then let W = (X-1)/(X+1)

Then DLOG(X) = W * SUM (C[i] * W**(2*i))

Otherwise, let W = (Y - DSQRT(2)/2)/(Y + DSQRT(2)/2)

Then DLOG(X) = A * DLOG(2) - 1/2 * DLOG(2) +
     W * SUM(C[i] * W**(2*i)
```

The coefficients are drawn from Hart #2662.[1]

The polynomial approximation used is of degree 6.

The relative error is less than or equal to 10**-9.9.

B.1.23 SIN—Real Floating Point Sine

SIN(X) is computed as:

```
Let Q = INTEGER(ABS(X)/(pi/2))
```

where:

Q = 0 for first quadrant
Q = 1 for second quadrant
Q = 2 for third quadrant
Q = 3 for fourth quadrant

```
Let Y = FRACTION((ABS(X)/(pi/2))
```

[1] Hart, *Computer Approximations*, p. 227.

If ABS(Y) < 2**-14, the sine is computed as:

```
SIN(X) = S * (pi/2)
   S = Y          if Q = 0
   S = 1-Y        if Q = 1
   S = -Y         if Q = 2
   S = Y-1        if Q = 3
```

For all other cases:

```
SIN(X) = P(Y*pi/2)          if Q = 0
SIN(X) = P((1-Y)*pi/2)      if Q = 1
SIN(X) = P(-Y*pi/2)         if Q = 2
SIN(X) = P((Y-1)*pi/2)      if Q = 3
```

where:

```
P = Y*SUM(C[i]*(Y**(2*i)))   for i = 0:4
```

The coefficients are taken from Hastings.[1]

The polynomial approximation used is of degree 4.

The relative error is less than or equal to 10**-8. The result is guaranteed to be within the closed interval -1.0 to +1.0.

B.1.24 DSIN—Double-Precision Floating Point Sine

DSIN(X) is computed as:

```
Let Q = INTEGER(ABS(X)/(pi/2))
```

where:

Q = 0 for first quadrant
Q = 1 for second quadrant
Q = 2 for third quadrant
Q = 3 for fourth quadrant

```
Let Y = FRACTION((ABS(X)/(pi/2))
```

[1] Hastings, C. et al., *Approximation for Digital Computers* (Princeton University Press, 1955), Sheet 16 (Part 2, p. 140).

If ABS(Y) < 2**-28, the sine is computed as:

```
    DSIN(X) = S * (pi/2)

       S = Y          if Q = 0
       S = 1-Y        if Q = 1
       S = -Y         if Q = 2
       S = Y-1        if Q = 3
```

For all other cases:

```
     DSIN(X) = P(Y*pi/2) <          if Q = 0
     DSIN(X) = P((1-Y)*pi/2)        if Q = 1
     DSIN(X) = P(-Y*pi/2)           if Q = 2
     DSIN(X) = P((Y-1)*pi/2)        if Q = 3
```

where:

```
P = Y*SUM(C[i]*(Y**(2*i)))   for i = 0:8
```

The coefficients are taken from Hastings.[1]

The polynomial approximation used is of degree 8.

The relative error is less than or equal to 10**-18.6. The result is guaranteed to be within the closed interval -1.0 to +1.0.

No loss of precision occurs if X < 2 * pi *256.

B.1.25 SQRT—Real Floating Point Square Root

SQRT(X) is computed as:

```
If X <= 0, an error is signaled. Therefore, let X = -X .

Let X = 2**K * F
```

where:

K is the exponential part of the floating point data.
F is the fractional part of the floating point data.

If K is even:

```
X = 2**(2P) * F
SQRT(X) = 2**P * SQRT (F)

   1/2 <= F < 1
```

[1] Hastings, C. et al., *Approximation for Digital Computers* (Princeton University Press, 1955), Sheet 16 (Part 2, p. 140).

where:

P = K/2.

If K is odd:

```
X = 2**(2P+1) * F = 2**(2P+2) * (F/2)
SQRT(X) = 2**(P+1) * SQRT(F/2)

   1/4 <= F/2 < 1/2
```

Let F′ = A*F + B, when K is even:

```
A = 0.453730314 (octal)
B = 0.327226214 (octal)
```

Let F′ = A*(F/2) + B, when K is odd:

```
A = 0.650117146 (octal)
B = 0.230170444 (octal)
```

Let K′ = P, when K is even
Let K′ = P+1, when K is odd

Let Y[0] = 2**K′ * F′ be a straight line approximation within the given interval using coefficients A and B, which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], two Newton-Raphson iterations are performed:

```
Y[n+1] = 1/2 * (Y[n] + X/Y[n])
```

The relative error is < 10**-8.

B.1.26 DSQRT—Double-Precision Floating Point Square Root

DSQRT(x) is computed as:

If X <= 0, an error is signaled. Therefore, let X = -X .
Let X = 2**K * F where:

K is the exponential part of the floating point data.
F is the fractional part of the floating data.

If K is even:

```
X = 2**(2P) * F
DSQRT(X) = 2**P * DSQRT (F)

   1/2 <= F < 1
```

If K is odd:

```
X = 2**(2P+1) * F = 2**(2P+2) * (F/2)
DSQRT(X) = 2**(P+1) * DSQRT(F/2)

   1/4 <= F/2 < 1/2
```

Let F′ = A*F + B, when K is even:

A = 0.453730314 (octal)
B = 0.327226214 (octal)

Let F′ = A*(F/2) + B, when K is odd:

A = 0.650117146 (octal)
B = 0.230170444 (octal)

Let K′ = P, when K is even.
Let K′ = P+1, when K is odd.

Let Y[0] = 2**K′ * F′ be a straight line approximation within the given interval using coefficients A and B, which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], three Newton-Raphson iterations are performed:

```
Y[n+1] = 1/2 * (Y[n] + X/Y[n])
```

The relative error is < 10**-17.

B.1.27 TAN—Real Floating Point Tangent

TAN(X) is computed as:

```
     SIN (X)/COS (X)

If COS(X) = 0 and SIN(X) > 0; error, return +
     If COS(X) = 0 and SIN(X) < 0; error, return -
```

where:

∞ is the largest representable number.

B.1.28 DTAN—Double-Precision Floating Point Tangent

DTAN(X) is computed as:

```
    DSIN (X)/DCOS(X)
```

```
If DCOS(X) = 0 and DSIN(X) > 0; error, return + If DCOS(X) = 0
and DSIN(X) < 0; error, return -
```

where:

∞ is the largest representable number.

B.2 Complex-Valued Procedures

B.2.1 CSQRT—Complex Square Root Function

CSQRT is computed as:

```
    ROOT = SQRT ((ABS (r) + CABS ((r,i))) / 2)

    Q = i / (2 * ROOT)
```

r	i	CSQRT ((r,i))
> =0	any	(ROOT, Q)
<0	> =0	(Q, ROOT)
<0	<0	(-Q, ROOT)

B.2.2 CSIN—Complex Sine

CSIN(Z) is computed as:

```
    (SIN(X) * cosh(Y), iCOS(X) * sinh(Y))
```

where:

```
    Z = X = iY
    cosh(Y) = (EXP(Y) + (1.0/EXP(Y)))/2
    sinh(Y) = (EXP(Y) - (1.0/EXP(Y)))/2
```

B.2.3 CCOS—Complex Cosine

CCOS(Z) is computed as:

```
(COS(X) * cosh(Y), i(-SIN(X) * sinh(Y))
```

where:

```
Z = X + iY
cosh(Y) = (EXP(Y) + (1.0/EXP(Y)))/2.0
sinh(Y) = (EXP(Y) - (1.0/EXP(Y)))/2.0
```

B.2.4 CLOG—Complex Logarithm

CLOG(Z) is computed as:

```
(ALOG(CABS(Z)), iATAN2(X,Y))
```

where:

```
Z = X + iY
```

B.2.5 CEXP—Complex Exponentia

CEXP(Z) is computed as:

```
EXP(X) * (COS(Y) +iSIN(Y))
```

where:

```
Z = X + iY
```

B.3 Random Number Generator

Two random number generators are available with FORTRAN-77: RANDOM and F77RAN. They are described in the following sections.

B.3.1 RANDOM—Uniform Pseudorandom Number Generator

This procedure is a general random number generator of the multiplicative congruential type. This means that it tends to be fast, but prone to nonrandom sequences when considering triples of numbers generated by this method. This procedure is called again to obtain the next pseudorandom number. The 32-bit seed is updated automatically. The result is a floating point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. There are no restrictions on the seed, although it should be initialized to different values on separate runs in order to obtain different random sequences. RANDOM uses the following to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 (MOD 2**32)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

RANDOM is invoked in one of three ways:

```
f = RAN(j)
f = RAN(i1,i2)
CALL RANDU(i1,i2,f)
```

where:

f is a real, floating point, random number
j is an INTEGER*4 seed
i1,i2 are INTEGER*2 seeds.

Notes:

1. Because the result is never 1.0, a simple way to get a uniform random integer selector is to multiply the value returned by the random number function by the number of cases. For example, if a uniform choice among five situations is to be made, then the following FORTRAN statement will work:

   ```
   GO TO (1,2,3,4,5),1 + IFIX(5.*RAN(ISEED))
   ```

   The explicit IFIX is necessary before adding 1 in order to avoid a possible rounding during the normalization after the addition of floating point numbers.

2. For further information on congruential generators and their limitations, see:

   G. Marsaglia, "Random Number Generation", in *The Encyclopedia of Computer Science,* ed., Anthony Ralston (Petrocelli/Charter, 1976), pp. 1192-1197.

B.3.2 F77RAN - Optional Uniform Pseudorandom Number Generator

This optional procedure is a general random number generator of the multiplicative congruential type. This procedure was the standard random number generator previous to Version 3.0 of PDP-11 FORTRAN and is included only for compatibility purposes as the file LB:[1,1]F77RAN.OBJ.

```
If I2=0, SEED = 2**16+3
otherwise, SEED = (2**16+3) * SEED (MOD 2**31)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

F77RAN is invoked in one of two ways:

```
    f= RAN (i1,i2)
    CALL RANDU (i1,i2,f)
```

where:

f is a real floating point, random number.
i1, i2 are INTEGER*2 seeds.

Appendix C

Diagnostic Messages

C.1 Diagnostic Message Overview

Diagnostic messages related to a FORTRAN–77 program can come from the compiler or from the OTS. The compiler detects syntax errors in a source program—such as unmatched parentheses, illegal characters, misspelled keywords, and missing or illegal parameters. The OTS reports errors that occur during execution.

C.2 Compiler Diagnostic Messages

Compiler diagnostic messages are generally self-explanatory; they specify the nature of an error and the action taken by the compiler. Besides reporting errors detected in source-program syntax, the compiler issues messages for errors such as I/O errors and stack overflow that involve the compiler itself.

C.2.1 Source Program Diagnostic Messages

The compiler distinguishes three classes of source-program errors, reported as follows:

F – Fatal errors that you must correct before a program can be compiled. If any F-class errors are reported in a compilation, the compiler produces no object file.

E – Errors that should be corrected. The program is not likely to run as intended with E-class errors; however, an object file is produced.

W – Warning messages that are issued for statements using nonstandard, though accepted, syntax and for statements corrected by the compiler. These statements may not have the intended result and you should check them before attempting execution. These messages are produced only when the warning switch (/WR) is set.

I – Information messages that although they do not call for corrective action, inform you that a correct FORTRAN-77 statement may have unexpected results. These messages are produced only when the warning switch (/WR) is set.

Errors detected during the initial phase of compiling appear immediately after the source line in which the error is presumed to have occurred; all other diagnostic messages appear immediately after the source listing.

Diagnostic messages issued by the compiler consist of two lines: The first line gives the error number and error message text; the second line contains a short section of the source line or the line number and/or the symbol that caused the diagnostic message.

One of the most frequent reasons for syntax errors, typing mistakes, can sometimes cause the compiler to give misleading diagnostic messages. You should avoid the following common typing mistakes:

- Missing commas or parentheses in complicated expressions or FORMAT statements.
- Particular instances of misspelled variable names. Because the compiler usually cannot detect these errors, execution may also be affected.
- Inadvertent line continuation marks, which can cause error messages for the preceding lines.
- Typing the uppercase letter O for the digit 0, or the reverse. If your terminal does not differentiate between the number and the letter, you may find it difficult to detect this error.

The presence of invalid ASCII characters in the source program can also cause misleading diagnostics. Nonprinting ASCII control characters except tab and form feed are not permitted in a FORTRAN-77 source program. If such control characters are detected, they are replaced by the question mark (?). However, because a question mark cannot occur in a FORTRAN-77 statement, this replacement can cause a syntax error.

Example C-1 shows the form of source-program diagnostic messages as they are displayed at your terminal in interactive mode. Example C-2 shows how these messages appear in listings.

Example C-1: Sample Diagnostic Messages (Terminal Format)

```
F77>COMERR=COMERR/NOF77

F77 -- ERROR 63-E Format item contains meaningless character
                  [RSTUVWXYZ'.I4,N] in module ERRCHK at line 5
F77 -- ERROR 85-W Name longer than 6 characters
                  [,LONGIDENTIFIER] in module ERRCHK at line 12
F77 -- ERROR 26-W No path to this statement
                  in module ERRCHK at line 17
F77 -- ERROR 10-E Multiple definition of a statement label, second
                  ignored [FORMAT] in module ERRCHK at line 20
F77 -- ERROR 50-F Undefined statement label
                  [ 102] in module ERRCHK
F77 -- 5 Errors   COMERR.FTN;3
```

The compiler diagnostic messages are as follows:

1 W Redundant continuation mark ignored

Explanation: A continuation mark is present where an initial line is required. The continuation mark is ignored.

2 W Invalid statement number ignored

Explanation: An improperly formed statement number is present in columns 1-5 of an initial line. The statement number has been ignored.

Example C–2: Sample Diagnostic Messages (Listing Format)

```
0001              PROGRAM ERRCHK
0002              PARAMETERS T=.TRUE.,F=.FALSE.
0003              INTEGER*4 TT,FF,I,J,II
0004              DATA TT,FF/T,F/
        C
0005    501       FORMAT('1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ',I4,M)
F77 -- ERROR 63-E Format item contains meaningless character
                  [RSTUVWXYZ',I4,M] in module ERRCHK at line 5

0006              OPEN(UNIT=1,NAME='FILE1.DAT',ACCESS='DIRECT',
                  1 RECORDSIZE=2)
0007              WRITE(1'1)TT,FF
        C
0008              TYPE 501,TT,FF
0009              TYPE 501,TT,FF
        C
0010              CALL SUBR
0011              READ(1,102)I,J,K
0012              READ(1,102,ERR=24)I,J,LONGIDENTIFIER
F77 -- ERROR 85-W Name longer than 6 characters
                  [,LONGIDENTIFIER] in module ERRCHK at line 12

0013    24        ASSIGN 92 TO X
0014              I=0
0015              J=3
0016              GO TO 24
0017              II=J/I
F77 -- ERROR 26-W No path to this statement
                  in module ERRCHK at line 17

0018    73        XX=Y/X
0019              TYPE 502,II,XX,ZZ
0020    501       FORMAT(2X,L2,2X,L2)
F77 -- ERROR 10-E Multiple definition of a statement label, second
                  ignored [FORMAT] in module ERRCHK at line 20

0021    502       FORMAT(2X,I5,2X,F,2X,F)
0022              CLOSE(UNIT=1,DISP='DELETE')
0023    92        STOP 'OK'
0024              END
F77 -- ERROR 50-F Undefined statement label
                  [ 102] in module ERRCHK

F77 -- 5 Errors COMERR.FTN;3
```

3 E Too many continuation lines, remainder ignored

Explanation: More continuation lines are present than were specified by the /CO:n qualifier. Up to 99 continuation lines are permitted. The default value is 19.

4 F Source line too long, compilation terminated

Explanation: A source line contains more than 88 characters. The compiler examines only the first 72 characters of a line.

5 E Statement out of order, statement ignored

Explanation: Statements must appear in the order specified in the *PDP-11 FORTRAN-77 Language Reference Manual.*

6 E Statement not valid in this program unit, statement ignored

Explanation: A program unit contains a statement that is not allowed; for example, an executable statement in a BLOCK DATA subprogram.

7 E Missing END statement, END is assumed

Explanation: An END statement is missing at the end of the last input file and has been inserted.

8 E Extra characters following a valid statement

Explanation: Extraneous text is present at the end of a syntactically valid statement. Check the entire statement for typing or syntax errors.

9 E Invalid initialization of variable not in COMMON

Explanation: An attempt was made in a BLOCK DATA subprogram to initialize a variable that is not in a COMMON block.

10 E Multiple definition of a statement label, second ignored

Explanation: Two or more statements have the same statement label. The first occurrence of the label is used.

11 F Compiler expression stack overflow

Explanation: An expression is too complex to be compiled. This error occurs in the following cases:

- An arithmetic or logical expression is too complex.
- There are too many actual arguments in a reference to a subprogram.
- There are too many parameters in an OPEN statement.

Explanation: The expression, subprogram reference, or OPEN statement must be simplified.

12 W Statement cannot terminate a DO loop

Explanation: The terminal statement of a DO loop cannot be a GO TO, arithmetic IF, RETURN, DO, or END statement.

13 E Count of Hollerith or Radix50 constant too large, reduced

Explanation: The integer count preceding H or R specifies more characters than remain in the source statement.

14 E Missing apostrophe in character constant

Explanation: A character constant must be enclosed by apostrophes.

15 F Missing variable or subprogram name

Explanation: A required variable name or subprogram name was not found.

16 E Multiple declaration of data type for variable, first used

Explanation: A variable cannot appear in more than one type declaration statement. The first type declaration is used.

17 E Constant in format item out of range

Explanation: A numeric value in a FORMAT statement exceeds the allowable range. Refer to the *PDP-11 FORTRAN-77 Language Reference Manual.*

18 E Invalid repeat count in DATA statement, count ignored

Explanation: The repeat count in a DATA statement is not an unsigned nonzero integer constant. It has been ignored.

19 F Missing constant

Explanation: A required constant was not found.

20 F Missing variable or constant

Explanation: An expression, or a term of an expression, has been omitted. Examples:

```
WRITE ( )
DIST = * TIME
```

21 F Missing operator or delimiter symbol

Explanation: Two terms of an expression are not separated by an operator, or a punctuation mark (such as a comma) has been omitted. Examples:

```
CIRCUM = 3.14  DIAM
```

22 F Multiple declaration of name

Explanation: A name appears in two or more inconsistent declaration statements.

23 E Syntax error in IMPLICIT statement

Explanation: Improper syntax was used in an IMPLICIT statement. Refer to the *PDP–11 FORTRAN–77 Language Reference Manual.*

24 E More than 7 dimensions specified, remainder ignored

Explanation: An array may have up to seven dimensions.

25 F Non-constant subscript where constant required

Explanation: In the DATA and EQUIVALENCE statements, subscript expressions must be constant.

26 W No path to this statement

Explanation: Program control cannot reach the statement. The statement is deleted.

27 E Adjustable array bounds must be dummy arguments or in COMMON

Explanation: Variables specified in dimension declarator expressions must either be subprogram dummy arguments or appear in COMMON.

28 E Overflow while converting constant or constant expression

Explanation: The specified value of a constant is too large or too small to be represented.

29 E Inconsistent usage of statement label

Explanation: Labels of executable statements have been confused with labels of FORMAT statements.

30 E Missing exponent after E or D

Explanation: A floating point constant is specified in E or D notation, but the exponent has been omitted.

31 E Invalid character used in hex, octal, or Radix–50 constant

Explanation: A character used was invalid. It must be one of the following:

- The valid Radix–50 characters are the letters A-Z, the digits 0-9, the dollar sign, the period, and the space. A space is substituted for the invalid character.
- The valid hexadecimal characters are 0-9, A-F, a-f.
- The valid octal characters are 0-7.

32 F Program storage requirements exceed addressable memory

Explanation: The storage space allocated to the variables and arrays of the program unit exceeds the addressing range of the PDP–11.

33 F Variable inconsistently equivalenced to itself

Explanation: The EQUIVALENCE statements of the program specify inconsistent relationships among variables and array elements. Example:

```
EQUIVALENCE (A(1), A(2))
```

34 F Undimensioned array or function definition out of order

Explanation: Either a statement function definition has been found among executable statements, or an assignment statement has been detected that involves an array for which dimension information has not been given.

35 F Format specifier in error

Explanation: The format specifier in an I/O statement is invalid. It must be one of the following:

- Label of a FORMAT statement
- * (list-directed)
- A run-time format specifier: variable, array, or array element
- Character constant containing a valid FORMAT specification

36 F Subscript or substring expression value out of bounds

Explanation: An array element has been referenced which is not within the specified dimension bounds.

37 F Invalid equivalence of two variables in COMMON

Explanation: Variables in COMMON cannot be equivalenced to each other.

38 F EQUIVALENCE statement incorrectly expands a COMMON block

Explanation: A COMMON block cannot be extended beyond its beginning by an EQUIVALENCE statement.

39 E Allocation begins on a byte boundary

Explanation: A non-BYTE quantity has been allocated to an odd byte boundary.

40 F Adjustable array used in invalid context

Explanation: A reference is made to an adjustable array in a context where such a reference is not allowed.

41 F Subscripted reference to non-array variable

Explanation: A variable that is not defined as an array cannot appear with subscripts.

42 F Number of subscripts does not match array declaration

Explanation: More or fewer dimensions are referenced than were declared for the array.

43 F Incorrect length modifier in type declaration

Explanation: The length specified in a type declaration statement is not compatible with the data type specified. Example:

```
INTEGER PIPES*8
```

44 F Syntax error in INCLUDE file specification

Explanation: The file name string is not acceptable (invalid syntax, invalid qualifier, undefined device, and so forth).

45 E Missing separator between format items

Explanation: A comma or other separator character has been omitted between fields in a FORMAT statement.

46 E Zero-length string

Explanation: The length specification of a character, Hollerith, or Radix–50 constant must be nonzero.

47 F Missing statement label

Explanation: A statement-label reference is not present where one is required.

48 F Missing keyword

Explanation: A keyword, such as TO, is omitted from a statement such as ASSIGN 10 TO I.

49 F Non-integer expression where integer value required

Explanation: An expression required to be of type INTEGER is of another data type.

50 F Undefined statement label

Explanation: A reference is made to a statement label that is not defined in the program unit.

51 E Number of names exceeds number of values in DATA statement

Explanation: The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. The remaining variables and/or array elements are not initialized.

52 E Number of values exceeds number of names in DATA statement

Explanation: The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. The remaining constant values are ignored.

53 F Statement cannot appear in logical IF statement

Explanation: The statement contained in a logical IF must not be a DO, logical IF, or END statement.

54 F Unclosed DO loops or block IF

Explanation: The terminal statement of a DO loop or the ENDIF statement of an IF block was not found.

55 W Assignment to DO variable within loop

Explanation: The control variable of a DO loop has been assigned a value within the loop.

56 F Variable name, constant, or expression invalid in this context

Explanation: A quantity has been incorrectly used: for example, the name of a subprogram where an arithmetic expression is required.

57 F Operation not permissible on these data types

Explanation: An invalid operation, such as .AND. on two real variables, is specified.

58 F Left side of assignment must be variable or array element

Explanation: The symbolic name to which the value of an expression is assigned must be a variable or array element.

59 F Syntax error in I/O list

Explanation: Improper syntax was detected in an I/O list.

60 E Constant size exceeds variable size in DATA statement

Explanation: The size of a constant in a DATA statement is greater than that of its corresponding variable.

61 E String constant truncated to maximum length

Explanation: The maximum length of a Hollerith constant or character constant is 255 characters; of a Radix-50 constant, 12.

62 E Lower bound greater than upper bound in array declaration

Explanation: The upper bound of a dimension must be greater than or equal to the lower bound.

63 E Format item contains meaningless character

Explanation: An invalid character or a syntax error is present in a FORMAT statement.

64 E Format item cannot be signed

Explanation: A signed constant is valid only with the P format code.

65 E Unbalanced parentheses in format list

Explanation: The number of right parentheses does not match the number of left parentheses.

66 E Missing number in format list

Explanation: There is a missing number in the format list. Example:

```
FORMAT (F6.)
```

67 E Extra number in format list

Explanation: There is an extra number in the format list. Example:

```
FORMAT (I4,3)
```

68 E Extra comma in format list

Explanation: There is an extra comma in the format list. Example:

```
FORMAT (I4,)
```

69 E Format groups nested too deeply

Explanation: Too many parenthesized format groups have been nested. Formats can be nested to only eight levels.

70 E END= or ERR= specification given twice, first used

Explanation: Two instances of either END= or ERR= were found. Control is transferred to the location specified in the first occurrence.

71 F Invalid I/O specification for this type of I/O statement

Explanation: A syntax error is in the portion of an I/O statement preceding the I/O list.

72 E Arguments incompatible with function, assumed user supplied

Explanation: A function reference has been made using an intrinsic function name, but the argument list does not agree in order, number, or type with the intrinsic function requirements. The function is assumed to be supplied by you as an external function.

73 E ENTRY within DO loop or IF block statement ignored

Explanation: An ENTRY statement is not permitted within the range of a DO loop.

74 F Statement too complex

Explanation: The statement is too large to compile. It must be subdivided into several statements.

75 F Too many named COMMON blocks

Explanation: Reduce the number of named COMMON blocks.

76 F INCLUDE files nested too deeply

Explanation: Reduce the level of INCLUDE nesting or increase the number of continuation lines permitted. Each INCLUDE file requires space for approximately two continuation lines.

77 F Duplicated keyword in OPEN/CLOSE statement

Explanation: A keyword subparameter of the OPEN or CLOSE statement cannot be specified more than once.

78 F DO and IF statements nested too deeply

Explanation: DO loops and IF blocks cannot be nested more than 20 levels.

79 F DO or IF statements incorrectly nested

Explanation: The terminal statements of a nest of DO loops or IF blocks are incorrectly ordered, or a terminal statement precedes its DO or block IF statement

80 F UNIT= keyword missing in OPEN/CLOSE statement

Explanation: The UNIT= subparameter of the OPEN and CLOSE statement must be present.

81 E Letter mentioned twice in IMPLICIT statement, last used

Explanation: An initial letter has been given an implicit data type more than once. The last data type given is used.

82 F Incorrect keyword in CLOSE statement

Explanation: A subparameter that can be specified only in an OPEN statement has been specified in a CLOSE statement.

83 F Missing I/O list

Explanation: An I/O list is not present where one is required.

84 F Open failure on INCLUDE file

Explanation: The file specified could not be opened. Possibly the file specification is incorrect, the file does not exist, the volume is not mounted, or a protection violation occurred.

85 W Name longer than 6 characters

Explanation: A symbolic name has been truncated to six characters.

86 F Invalid virtual array usage

Explanation: A virtual array has been used in a context that is not permitted.

87 F Invalid key specification

Explanation: The key value in a keyed I/O statement must be a character constant, a BYTE array name, or an integer expression.

88 F Non-logical expression where logical value required

Explanation: An expression that must be of type LOGICAL is of another data type.

89 E Invalid control structure using ELSEIF, ELSE, or ENDIF

Explanation: The order of ELSEIF, ELSE, or ENDIF statement is incorrect.

ELSEIF, ELSE, and ENDIF statements cannot stand alone. ELSEIF and ELSE must be preceded by either a block IF statement or an ELSEIF statement. ENDIF must be preceded by either a block IF, ELSEIF, or ELSE statement. Examples:

```
        DO 10 I=1,10
            J=J+I
            ELSEIF (J.LE.K)THEN
    ERROR: ELSE IF preceded by a DO statement.

        IF (J.LT.K)THEN
           J=I+J
        ELSE
           J=I-J
        ELSEIF (J.EQ.K)THEN
        ENDIF

    ERROR: ELSEIF preceded by an ELSE statement.
```

90 F Name previously used with conflicting data type

Explanation: A data type is assigned to a name that has already been used in a context that required a different data type.

91 E Character name incorrectly initialized with numeric value

Explanation: Character data with a length greater than 1 is initialized with a numeric value in a data statement. Example:

```
Character *4 A
DATA A/14/
```

92 E Substring reference used in invalid context

Explanation: A substring reference to a variable or array that is not of data type CHARACTER has been detected. Example:

```
REAL X(10)
Y=X(J:K)
```

93 F Character substring limits out of order

Explanation: The first character position of a substring expression is greater than the last character position. Example:

```
C(5:3)
```

94 W Mixed numeric and character elements in COMMON

Explanation: A COMMON block must not contain both numeric and character data.

95 E Invalid ASSOCIATEVARIABLE specification

Explanation: An ASSOCIATEVARIABLE specification in an OPEN or DEFINE FILE statement is a dummy argument or an array element.

96 E ENTRY dummy variable previously used in executable statement

Explanation: The dummy arguments of an ENTRY statement must not have been used previously in an executable program in the same program unit.

97 E Invalid use of intrinsic function as actual argument

Explanation: A generic intrinsic function name was used as an actual argument.

98 E Name used in INTRINSIC statement is not an intrinsic function

Explanation: A function name that appears in an INTRINSIC statement is not an intrinsic function.

99 E Non-blank characters truncated in string constant

Explanation: A character or Hollerith constant was converted to a data type that was not large enough to contain all significant digits.

100 E Non-zero digits truncated in hex or octal constant

Explanation: An octal or hexadecimal constant was converted to a data type that was not large enough to contain all significant digits.

101 W Mixed numeric and character elements in EQUIVALENCE

Explanation: Numeric and character variable and array elements cannot be equivalenced to each other.

102 F Arithmetic expression where character value required

Explanation: An expression that must be of data type CHARACTER was another data type.

103 F Assumed size array name used in invalid context

Explanation: An assumed size array name was used where the size of the array was also required, for example, in an I/O list.

104 F Character expression where arithmetic value required

Explanation: An expression that must be arithmetic (integer, real, logical, or complex) is of data type character.

105 F Function or entry name not numeric

Explanation: Functions of data type character are not allowed.

106 I Default STATUS='UNKNOWN' used in OPEN statement

Explanation: The OPEN statement default STATUS='UNKNOWN' may cause an old file to be modified inadvertently.

107 I Extension to FORTRAN–77: tab indentation or lowercase source

Explanation: The use of tab characters or lowercase source letters in the source code is an extension to the ANSI FORTRAN standard.

108 I Extension to FORTRAN–77: non-standard comment

Explanation: The ANSI FORTRAN standard allows only the characters C and * to begin a comment line; D, d, and ! are extensions to the standard.

109 I Extension to FORTRAN–77: non-standard statement type

Explanation: A nonstandard statement type was used. See Appendix G.

110 I Extension to FORTRAN–77: non-standard lexical item

Explanation: One of the following nonstandard lexical items was used:

- The single-quote form of record specifier in a direct access I/O statement
- A variable format expression

111 I Extension to FORTRAN–77: non-standard operator

Explanation: The operator .XOR. is an extension to the ANSI FORTRAN standard. The standard form of .XOR. is .NEQV..

112 I Extension to FORTRAN–77: non-standard keyword

Explanation: A nonstandard keyword was used. See Appendix G.

113 I Extension to FORTRAN-77: non-standard constant

Explanation: The following constant forms are extensions to the ANSI FORTRAN standard:

Hollerith	nH . . . .
Typeless	'xxxx'X or 'oooo'O
Octal	"oooo or Ooooo
Hexadecimal	Zxxxx
Radix-50	nR . . . .
Complex with PARAMETER components	

114 I Extension to FORTRAN-77: non-standard data type specification

Explanation: The following data type specifications are extensions to the ANSI FORTRAN standard. The acceptable equivalent in the standard language is given where appropriate. This message is issued when these type specifications are used in the IMPLICIT statement or in a numeric type statement that contains a data type length override.

Extension	Standard
BYTE	
LOGICAL*1	
LOGICAL*2	Logical
LOGICAL*4	Logical (only with /I4)
INTEGER*2	Integer
INTEGER*4	Integer (only with /I4)
REAL*4	Real
REAL*8	Double Precision
COMPLEX*8	Complex

115 I Extension to FORTRAN-77: non-standard syntax

Explanation: One of the following syntax extensions was specified:

PARAMETER name = value	No parentheses around name = value.
IMPLICIT type letter	See Section G.2.1 for explanation.
CALL name (arg1,,arg3)	Null actual argument.
READ (. . .),iolist	Comma between I/O control and element lists.
e1 * -e2	Two consecutive operators.

116 I Extension to FORTRAN-77: non-standard FORMAT statement item

Explanation: The following format field descriptors are extensions to the ANSI FORTRAN standard:

S, Q, O, Z	All forms
(A,L,I,F,E,G,D)	Default field width forms
P	Without scale factor

117 F Untyped name, must be explicitly typed

Explanation: An IMPLICIT NONE statement is included in the program unit, and a symbolic name has not been assigned a data type.

118 I Extension to FORTRAN-77: non-standard source line length

Explanation: The ANSI FORTRAN standard allows source lines of up to 72 characters; source lines exceeding 72 characters are extensions to the standard.

C.2.2 Compiler — Fatal Diagnostic Messages

Certain error conditions can occur during compilation that are so severe that the compilation must be terminated immediately. The following messages report such errors. Included are hardware error conditions, conditions that may require you to modify the source program, and conditions that are the result of software errors.

```
F77 -- FATAL 01 * Open error on work file (LUN 6)
F77 -- FATAL 02 * Open error on temp file (LUN 7,8)
```

During the compilation process, FORTRAN-77 creates a temporary work file and zero, one, or two temporary scratch files; the compiler was unable to open these required files. Possibly the volume was not mounted, space was not available on the volume, or a protection violation occurred.

```
F77 -- FATAL 03 * I/O error on work file (LUN 6)
F77 -- FATAL 04 * I/O error on temp file (LUN 7,8)
F77 -- FATAL 05 * I/O error on source file
F77 -- FATAL 06 * I/O error on object file
F77 -- FATAL 07 * I/O error on listing file
```

I/O errors report either hardware I/O errors or such software error conditions as an attempt to write on a write-protected volume.

```
F77 -- FATAL 08 * Compiler dynamic memory overflow
```

Reduce the number of continuation lines allowed, reduce the INCLUDE file nesting depth, run in a larger partition, or rebuild or reinstall the compiler with a larger dynamic memory area.

```
F77 -- FATAL 09 * Compiler virtual memory overflow
```

A single program unit is too large to be compiled. Specify /WF:3 or divide the program into smaller units.

```
F77 -- FATAL 10 * Compiler internal consistency check
```

An internal consistency check has failed. This error should be reported to DIGITAL in a Software Performance Report; see Appendix G.

```
F77 -- FATAL 11 * Compiler control stack overflow
```

The compiler's control stack overflowed. Simplifying the source program will correct the problem.

C.2.3 Compiler Limits

There are limits to the size and complexity of a single FORTRAN-77 program unit. There are also limits on the complexity of FORTRAN statements. In some cases, the limits are readily described; see Table C-1. In other cases, however, the limits are not so easily defined.

For example, the compiler uses external work files to store the symbol table and a compressed representation of the source program. The /WF qualifier controls the number of work files: The maximum is 3, which provides space for approximately 2000 or more lines of source code in a typical FORTRAN program unit. If you run out of work file space, compiler fatal error 9 occurs.

In some cases, you can adjust the limits by relinking the compiler and modifying the limits to suit your needs. Table C–1 shows two values for such limits, in the form m(n), where m is the default limit and n is the maximum. Limits for which only one value is shown are not adjustable. Consult the *PDP–11 FORTRAN–77 Installation Guide* for information about modifying compiler limits and relinking the compiler.

Table C–1: Compiler Limits

Language Element	Limit
DO nesting	20 (many)
Block IF nesting	20 (32)
Actual arguments per CALL or function reference	32 (120)
OPEN statement keyword	16 (60)
Named COMMON blocks	45 (250)
Saved named COMMON blocks	45 (128)
Format group nesting	8
Labels in computed or assigned GOTO list	250
Parentheses nesting in expressions	24 (many)
INCLUDE file nesting	10
Continuation line	99
FORTRAN source line length	88 characters (132)
Symbolic name length	6 characters

Table C-1 (Cont.): Compiler Limits

Language Element	Limit
Constants:	
Character	255 characters
Hollerith	255 characters
Radix-50	12 characters
Array dimension	7

C.3 Object Time System Diagnostic Messages

The following sections provide information on the formats and contents of OTS diagnostic messages, and a list of OTS error messages arranged by error code.

C.3.1 Object Time System Diagnostic Message Format

An OTS diagnostic message consists of several lines of information formatted as follows:

```
tsknam  --  [EXITING DUE TO] ERROR number
text
[AT PC = address]
[I/O:  ioerr ioerr1 unit filespec]
   IN    xxxxxx [AT [OR AFTER] yyy]
   FROM  xxxxxx [AT [OR AFTER] yyy]
   ...
   FROM  xxxxxx [AT [OR AFTER] yyy]
```

(In the above message prototype, fixed parts of the message are shown in uppercase letters and variable parts in lowercase letters.)

The variable parts of the message are:

tsknam The name of the task in which the error occurred.

number The error number.

text A 1-line description of the error.

The phrase "EXITING DUE TO" is included only when the error is causing program termination. If a program is terminated by the OTS, the termination status value is severe error.

If the OTS error results from one of the synchronous system traps or a Floating Point Processor trap, the program counter is shown in the line AT PC =. This line is produced only for errors numbered 3 through 14 and 72 through 75.

If the OTS error results from an I/O error condition detected by the file system, the line beginning I/O: is included.

ioerr — The primary error code; this value is the F.ERR value for the FCS-11 file system or the O$STS value for the RMS–11 file system.

ioerr1 — The secondary error code; this value is the F.ERR+1 value for FCS-11 or the O$STV value for RMS–11.

unit — The logical unit on which this error occurred.

filespec — The file name, file type, and version number of the file.

Next follows a traceback of the subprogram calling nest at the time of the error. Each line represents one level of subprogram call and shows

xxxxxx — The name of the subprogram.

The name of the main program is shown as .MAIN. unless a PROGRAM statement has been used. The name of a subprogram is the same as the name used in the SUBROUTINE, FUNCTION, or ENTRY statement. Statement functions, OTS system routines, and routines written in assembly language are not shown in the traceback.

A program unit compiled with the /TR:NONE switch in effect is not included in the traceback list.

yyy — The internal sequence number of the subprogram at which the error, call statement, or function reference occurred.

If a program unit is compiled with the /TR:ALL switch in effect, then the text AT yyy indicates the exact internal sequence number at which the error occurred.

If a program unit is compiled with the /TR:BLOCKS switch in effect, then the text AT OR AFTER yyy indicates that the error occurred in the block starting at sequence number yyy.

If a program unit is compiled with the /TR:NAMES option in effect, then no sequence information is available and no text or sequence number follows the routine name.

NOTE

With Floating Point Processor errors, it is possible for the internal sequence number shown in the first traceback line to be the sequence number of the next statement. This results from the asynchronous relationship between the central processor and the FPP, and occurs when the CPU starts execution of the next statement before the FPP error trap is initiated.

Example C–3 is a sample terminal listing of several object time system diagnostic messages.

Example C–3: Sample of Object Time System Diagnostic Messages

```
TTnn -- ERROR 37
Inconsistent record length
  IN    "ERRCHK" AT 00022

TTnn -- ERROR 34
Unit already open
  IN    "SUBR2" AT OR AFTER 00002
  FROM  "SUBR1"
  FROM  "ERRCHK" AT 00025

TTnn -- ERROR 64
Input conversion error
  IN    "ERRCHK" AT 00026

TTnn -- ERROR 24
End-of-file during read
FCS -10, 0 1 FILE1.DAT;1
  IN    "ERRCHK" AT 00028

TTnn -- ERROR 73
Floating zero divide
at PC = 024656
  IN    "ERRCHK" AT 00036

TTnn -- ERROR 84
Square root of negative value
  IN    "FUNC" AT 00002
  FROM  "ERRCHK" AT 00037

TTnn -- Exiting due to ERROR 29
No such file
FCS -26, 0 4 TEMPFILE.DAT
  IN    "ERRCHK" AT 00042
```

C.3.2 Object Time System Error Codes

The following messages result from severe run-time error conditions for which no error recovery is possible. Consult the operating system's Executive reference manual for details of what error conditions cause traps to the System Synchronous Trap Table entries cited.

1 Invalid error call

A TRAP instruction has been executed whose low byte is within the range used by the OTS for error reporting but for which no error condition is defined.

2 Task initialization failure

Task startup has failed for one of the following reasons:

- The directive to initialize synchronous system trap handling (SVTK$S) has returned an error indication.
- The executive directive to enable the FPP asynchronous trap (SFPA$S) has returned an error indication. This error will be returned if the task was task-built with /-FP specified, or if the hardware configuration does not contain an FPP.
- The FCS-11 initialization call (FINIT$) or RMS–11 initialization call ($INITIF) has returned an error indication.

3 Odd address trap (SST0)

The program has made a word reference to an odd byte address.

4 Segment fault (SST1)

The program has referenced a nonexistent address, most likely due to a subscript value out of range on an array reference.

5 T-bit or BPT trap (SST2)

A trap has occurred as a result of the trace bit being set in the processor status word or of the execution of a BPT instruction.

6 IOT trap (SST3)

A trap has occurred as a result of the execution of an IOT instruction.

7 Reserved instruction trap (SST4)

The program has attempted to execute an illegal instruction.

8 Non-RSX EMT trap (SST5)

The program has executed an EMT instruction whose low byte is not in the range used by the operating system.

9 TRAP instruction trap (SST6)

A TRAP instruction has been executed whose low byte is outside the range used for OTS error messages.

10 PDP–11/40 FIS trap (SST7)

This message may result when an operating system which was generated for an 11/40 is used on another PDP–11 processor.

11 FPP hardware fault

The FPP Floating Exception Code (FEC) register contained the value 0 following an FPP interrupt. This is probably a hardware malfunction.

12 FPP illegal opcode trap

The FPP has detected an illegal floating point instruction.

13 FPP undefined variable trap

The FPP loaded an illegal value (-0.0). This trap should not occur since the OTS initialization routine does not enable this trap condition. A negative zero value should never be produced by any FORTRAN operation.

14 FPP maintenance trap

The FPP Floating Exception Code register contained the value 14 (octal) following a FPP interrupt. This is probably a hardware malfunction.

The following messages result from errors related to the file system:

20 REWIND error

An error condition was detected by FCS-11 during the .POINT operation or by RMS–11 during the $REWIND operation used to position at the beginning of a file.

21 Duplicate file specifications

Multiple attempts to specify file attributes have been attempted, without an intervening close operation, by one of the following:

- DEFINEFILE followed by DEFINEFILE
- DEFINEFILE, CALL ASSIGN, or CALL FDBSET followed by an OPEN statement.

22 Input record too long

A record too large to fit into the user record buffer has been read. Rebuild the task using a larger Task Builder MAXBUF value (see Section 1.2.5.2) and specify a larger RECL for the file.

23 BACKSPACE error

One of the following errors has occurred:

- BACKSPACE was attempted on a relative or indexed file or a file opened for append access (see Section 2.3).
- FCS-11 or RMS-11 has detected an error condition while rewinding the file.
- FCS-11 or RMS-11 has detected an error condition while reading forward to the desired record.

24 End-of-file during read

Either an end-file record produced by the ENDFILE statement or an end-of-file condition has been encountered during a READ statement, and no END= transfer specification was provided.

25 Record number outside range

A direct access I/O statement has specified a record number outside the range specified in a DEFINEFILE statement or in the MAXREC keyword of the OPEN statement.

26 Access mode not specified

The access mode of an I/O statement was inconsistent with the access specified by a DEFINEFILE or OPEN statement for the logical unit.

27 More than one record in I/O statement

An attempt was made to process more than a single record in a REWRITE statement or in an ENCODE or DECODE statement.

28 Close error

An error condition has been detected during the close, delete, or print operation of an attempt to close a file.

29 No such file

A file with the specified name could not be found during an open operation.

30 Open failure

FCS-11 or RMS-11 has detected an error condition during an open operation. (This message is used when the error condition is not one of the more common conditions for which specific error messages are provided.)

31 Mixed file access modes

An attempt was made to use both formatted and unformatted operations, or both sequential and direct access operations, on the same unit.

32 Invalid logical unit number

- A logical unit number was used that is greater than 99, less than 0, or outside the range specified by the Task Builder UNITS option (see Section 1.2.5.2).
- A logical unit number of 0 was not mapped to a valid logical unit number (1-99) specified by the Task Builder option GBLPAT (see Section 2.1.3).

33 ENDFILE error

An end-file record may not be written to a direct access file, a relative file, an indexed file, or an unformatted file that does not contain segmented records.

34 Unit already open

An OPEN statement or DEFINEFILE statement was attempted that specified a logical unit already opened for input/output.

35 Segmented record format error

Invalid segmented record control data was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or written by a language other than FORTRAN.

36 Attempt to access non-existent record

One of the following conditions has occurred:

- A nonexistent record was specified in a direct access READ or FIND statement to a relative file. The nonexistent record might have been deleted or never written.
- A record located beyond the end-of-file was specified in a direct access READ or FIND statement.
- No record matches the key value of a keyed access READ statement.

37 Inconsistent record length

An invalid or inconsistent record length specification occurred for one of the following reasons:

- The record length specified is too large to fit in the user record buffer. Rebuild the task with a larger Task Builder MAXBUF value.
- The record length specified does not match the record length attribute of an existing fixed-length file.
- The record length specification was omitted when an attempt was made to create a relative file or a file with fixed-length records.

38 Error during write

FCS-11 or RMS–11 has detected an error condition during execution of a WRITE statement.

39 Error during read

FCS-11 or RMS–11 has detected an error condition during execution of a READ statement.

40 Recursive I/O operation

An expression in the I/O list of an I/O statement has caused initiation of another I/O operation. This can happen if a function that performs I/O is referenced in an expression in an I/O list.

41 No buffer room

There is not enough free memory left in the OTS buffer area to set up required I/O control blocks and buffers. Rebuild the task with a larger Task Builder ACTFIL option (see Section 1.2.5.2). For RMS–11, rebuild the task with a larger EXTTSK value, or run the task with a larger task increment. For FSC, if the correct ACTFIL has been specified, see Section 5.6 for information on how to work around fragmentation of the $$FSR1 buffer area.

42 No such device

A file name specification has included an invalid device name or a device for which no handler task is loaded when an open operation is attempted.

43 File name specification error

The file name string used in a CALL ASSIGN or OPEN statement is syntactically invalid, contains a qualifier specification, references an undefined device, or is otherwise not acceptable to the operating system.

44 Inconsistent record type

The RECORDTYPE specification does not match the record type of an existing file.

45 Keyword value error in OPEN statement

An OPEN statement keyword that requires a value has an illegal value. The following values are accepted:

BLOCKSIZE:	0	to	32767
EXTENDSIZE:	-32768	to	32767
INITIALSIZE:	-32768	to	32767
MAXREC:	0	to	2**31-1
BUFFERCOUNT:	0	to	127
RECL: up to	32766	for	sequential organization
	16360	for	relative or indexed organization
	9999	for	magnetic tape

46 Inconsistent OPEN/CLOSE parameters

The specifications in an OPEN and/or subsequent CLOSE statement have incorrectly specified one or more of the following:

- A 'NEW' or 'SCRATCH' file which is 'READ-ONLY'
- 'APPEND' to a 'NEW', 'SCRATCH', or 'READONLY' file
- 'SAVE' or 'PRINT' on a 'SCRATCH' file
- 'DELETE' or 'PRINT' on a 'READONLY' file.

47 Write to read-only file

A write operation has been attempted to a file which was declared to be READONLY.

48 Unsupported I/O operation

An I/O operation (such as direct or keyed access) has been specified which is not supported by the OTS being used.

49 Invalid key specification

A key specification value, such as position, size, or key-of-reference number, was invalid in an OPEN or READ statement. Examples:

```
OPEN (UNIT=1,RECL=40,KEY=(200:220))
```

or

```
READ (UNIT=1,KEY='ABCD',KEYID=-1)
```

50 Inconsistent key change or duplicate key value

A keyed WRITE or REWRITE statement specified an invalid key value for one or more of the following reasons:

- A key value changed that is not allowed to change.
- A key value duplicated the key value of another record, but duplicate key values are not permitted.

51 Inconsistent file organization

The value of the ORGANIZATION keyword in an OPEN statement does not match the organization of the existing file being opened.

52 Specified record locked

The record specified by an I/O statement was locked by another program or another logical unit within your program.

53 No current record

A REWRITE or sequential DELETE statement was executed but no current record was defined. Sequential REWRITE and DELETE statements must be preceded by a successful READ statement.

54 REWRITE error

An error occurred during execution of a REWRITE statement, or an attempt was made to rewrite a record in a sequential or relative file.

55 DELETE error

An error occurred during execution of a DELETE statement, or an attempt was made to delete a record from a sequential file.

56 UNLOCK error

An error occurred during execution of an UNLOCK statement.

57 FIND error

An error occurred during execution of a FIND statement.

The following messages result from errors related to transmitting data between a FORTRAN-77 program and an internal record:

59 List-directed I/O syntax error

The data in a list-directed input record has an invalid format or the type of the constant is incompatible with the corresponding variable. The value of the variable is unchanged.

60 Infinite format loop

The format associated with an I/O statement that includes an I/O list has *no field descriptors to use in transferring those variables.* For example:

```
     WRITE (I,1)X
1    FORMAT (' X=')
```

61 Format/variable-type mismatch

An attempt was made to input or output a real variable with an integer field descriptor (I or L), or an integer or logical variable with a real field descriptor (D, E, F, or G). The data type of the value is ignored, and the value is processed as if it were of the correct data type.

62 Syntax error in format

A syntax error was encountered while the OTS was processing a format stored in an array.

63 *Output conversion error*

During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. The field is filled with asterisks (*).

64 Input conversion error

During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable is set to zero.

65 Format too big for 'FMTBUF'

The OTS has run out of memory while scanning an array format that was generated at run time. The default internal format buffer length is 64 bytes. You can increase this length by using the Task Builder FMTBUF option (see Section 1.2.5.2).

66 Output statement overflows record

An output operation has specified a record that exceeds the maximum record size specified. The maximum record length is specified by the DEFINEFILE statement, by the RECL keyword of the OPEN statement, or by the record length attribute of an existing file. See Section F.1.7.

67 Record too small for I/O list

A READ statement has attempted to input more data than existed in the record being read. For example, the I/O list might have too many elements.

68 Variable format expression value error

The value of a variable format expression is not within the range acceptable for its intended use: for example, a field width that is less than or equal to zero. A value of 1 is used.

The following messages result from arithmetic overflow and underflow conditions:

70 Integer overflow

During an arithmetic operation, an integer's value has exceeded INTEGER*4 range. (Note: Overflow of INTEGER*2 range involving INTEGER*2 variables is not detected.)

71 Integer zero divide

During an integer mode arithmetic operation, an attempt was made to divide by zero. (Note: A zero-divide operation involving INTEGER*2 variables is rarely detected.)

72 Floating overflow

During an arithmetic operation, a real value has exceeded the largest representable real number. The result of the operation is set to zero.

73 Floating zero divide

During a real mode arithmetic operation, an attempt was made to divide by zero. The result of the operation is set to zero.

74 Floating underflow

During an arithmetic operation, a real value has become less than the smallest representable real number and has been replaced with a value of zero.

75 FPP floating to integer conversion overflow

The conversion of a floating point value to an integer has resulted in a value that overflows the range representable in an integer. The result of the operation is zero.

The following messages result from incorrect calls to FORTRAN-77 supplied functions or subprograms:

80 Wrong number of arguments

One of the FORTRAN library functions or system subroutines has been called with an improper number of arguments (see Table 4–1 or Appendix D).

81 Invalid argument

One of the FORTRAN library functions or system subroutines has detected an invalid argument value. (see Table 4–1 or Appendix D).

82 Undefined exponentiation

An exponentiation (for example, 0.**0.) has been attempted that is mathematically undefined. The result returned is zero.

83 Logarithm of zero or negative value

An attempt was made to take the logarithm of zero or a negative number. The result returned is zero.

84 Square root of negative value

An argument required the evaluation of the square root of a negative value. The square root of the absolute value is computed and returned.

The following miscellaneous errors are detected.

91 Computed GOTO out of range

The integer variable or expression in a computed GO TO statement was less than 1 or greater than the number of statement label references in the list. Control is transferred to the next executable statement.

92 Assigned label not in list

An assigned GOTO has been executed in which the label assigned to the variable is not one of the labels in the list. Control is transferred to the next executable statement.

93 Adjustable array dimension error

Upon entry to a subprogram, the evaluation of dimensioning information has detected an array in which one of the following occurs:

- An upper dimension bound is less than a lower dimension bound
- The dimensions imply an array which exceeds the addressable memory.

94 Array reference outside array

An array reference has been detected that is outside the array as described by the array declarator. Execution continues. (This checking is performed only for program units compiled with the /CK switch in effect.)

95 Incompatible FORTRAN object module in task

An object module produced by another PDP–11 FORTRAN compiler has been linked with a FORTRAN–77 task (see Section 1.2.5.1).

96 Missing format conversion routine

- A format conversion code has been used for which the corresponding conversion routine is not loaded (see Section 3.4).
- An F4P V3 object file that uses octal format may have been task-built with the F77 OTS. Re-task-build with the option GBLREF = ZCI$, or recompile, with the F77 V4 compiler, the modules that use octal format; then task-build as usual.

97 FTN FORTRAN error call

The error-reporting subroutine entry used by the FTN FORTRAN system has been called. Possibly an FTN object module or FTN-dependent service subroutine has been included in the task.

98 User requested traceback

A user-supplied MACRO–11 subprogram has requested a subroutine calling nest-traceback display. Execution continues.

The following messages result from incorrect calls to system directive subroutines:

100 Directive: Missing argument(s)

A call to a system directive subroutine was made in which one or more of the arguments required for directive execution was not given.

101 Directive: Invalid event flag number

A call to a system directive subroutine was made in which the argument used for event flag specification was not in the valid range (1 to 64).

The following messages result from incorrect usage of virtual arrays:

111 Virtual array initialization failure

The mapped array area could not be initialized. The operating system does not support the memory management directives required, or no memory management registers are available for use.

112 Virtual array mapping error

A virtual-array address was invalid, probably due to a subscript out of bounds. Execution continues.

C.4 Operating System and File System Error Codes

The following sections list the error-code names and values for operating system and file system errors that occur during run-time.

C.4.1 Operating System Error Codes

Standard operating system error codes returned during run time by directives in the Directive Status Word are as follows.

IE.UPN	177777	-01.	INSUFFICIENT DYNAMIC STORAGE
IE.INS	177776	-02.	SPECIFIED TASK NOT INSTALLED
IE.PTS	177775	-03.	PARTITION TOO SMALL FOR TASK
IE.UNS	177774	-04.	INSUFFICIENT DYNAMIC STORAGE FOR SEND
IE.ULN	177773	-05.	UN-ASSIGNED LUN
IE.HWR	177772	-06.	DEVICE HANDLER NOT RESIDENT
IE.ACT	177771	-07.	TASK NOT ACTIVE
IE.ITS	177770	-08.	DIRECTIVE INCONSISTENT WITH TASK STATE
IE.FIX	177767	-09.	TASK ALREADY FIXED/UNFIXED
IE.CKP	177766	-10.	ISSUING TASK NOT CHECKPOINTABLE
IE.TCH	177765	-11.	TASK IS CHECKPOINTABLE
IE.RBS	177761	-15.	RECEIVE BUFFER IS TOO SMALL
IE.PRI	177760	-16.	PRIVILEGE VIOLATION
IE.RSU	177757	-17.	RESOURCE IN USE
IE.NSW	177756	-18.	NO SWAP SPACE AVAILABLE
IE.ILV	177755	-19.	ILLEGAL VECTOR SPECIFIED
IE.AST	177660	-80.	DIRECTIVE ISSUED/NOT ISSUED FROM AST
IE.MAP	177657	-81.	ILLEGAL MAPPING SPECIFIED
IE.IOP	177655	-83.	WINDOW HAS I/O IN PROGRESS
IE.ALG	177654	-84.	ALIGNMENT ERROR
IE.WOV	177653	-85.	ADDRESS WINDOW ALLOCATION OVERFLOW
IE.NVR	177652	-86.	INVALID REGION ID
IE.NVW	177651	-87.	INVALID ADDRESS WINDOW ID
IE.ITP	177650	-88.	INVALID TI PARAMETER
IE.IBS	177647	-89.	INVALID SEND BUFFER SIZE (.GT. 255.)
IE.LNL	177646	-90.	LUN LOCKED IN USE
IE.IUI	177645	-91.	INVALID UIC
IE.IDU	177644	-92.	INVALID DEVICE OR UNIT
IE.ITI	177643	-93.	INVALID TIME PARAMETERS
IE.PNS	177642	-94.	PARTITION/REGION NOT IN SYSTEM
IE.IPR	177641	-95.	INVALID PRIORITY (.GT. 250.)
IE.ILU	177640	-96.	INVALID LUN
IE.IEF	177637	-97.	INVALID EVENT FLAG (.GT. 64.)
IE.ADP	177636	-98.	PART OF DPB OUT OF USER'S SPACE
IE.SDP	177635	-99.	DIC OR DPB SIZE INVALID

C.4.2 Summary of FCS-11 Error Codes

Directive error codes are returned during run time to FCS in the Directive Status Word. FCS returns these codes in byte F.ERR of the File Descriptor Block. Byte F.ERR+1 in the FDB distinguishes Directive error codes from the overlapping codes from within the file system by showing negative values for the Directive error codes.

File system error codes are returned by FCS-11 in byte F.ERR in the File Descriptor Block. Byte F.ERR+1 is 0 if F.ERR contains a file system error code.

IE.BAD	177777	-01.	BAD PARAMETERS
IE.IFC	177776	-02.	INVALID FUNCTION CODE
IE.DNR	177775	-03.	DEVICE NOT READY
IE.VER	177774	-04.	PARITY ERROR ON DEVICE
IE.ONP	177773	-05.	HARDWARE OPTION NOT PRESENT
IE.SPC	177772	-06.	ILLEGAL USER BUFFER
IE.DNA	177771	-07.	DEVICE NOT ATTACHED
IE.DAA	177770	-08.	DEVICE ALREADY ATTACHED
IE.DUN	177767	-09.	DEVICE NOT ATTACHABLE
IE.EOF	177766	-10.	END OF FILE DETECTED
IE.EOV	177765	-11.	END OF VOLUME DETECTED
IE.WLK	177764	-12.	WRITE ATTEMPTED TO LOCKED UNIT
IE.DAO	177763	-13.	DATA OVERRUN
IE.SRE	177762	-14.	SEND/RECEIVE FAILURE
IE.ABO	177761	-15.	REQUEST TERMINATED
IE.PRI	177760	-16.	PRIVILEGE VIOLATION
IE.RSU	177757	-17.	SHARABLE RESOURCE IN USE
IE.OVR	177756	-18.	ILLEGAL OVERLAY REQUEST
IE.BYT	177755	-19.	ODD BYTE COUNT (OR VIRTUAL ADDRESS)
IE.BLK	177754	-20.	LOGICAL BLOCK NUMBER TOO LARGE
IE.MOD	177753	-21.	INVALID UDC MODULE #
IE.CON	177752	-22.	UDC CONNECT ERROR
IE.BBE	177710	-56.	BAD BLOCK ON DEVICE
IE.STK	177706	-58.	NOT ENOUGH STACK SPACE (FCS OR FCP)
IE.FHE	177705	-59.	FATAL HARDWARE ERROR ON DEVICE
IE.EOT	177702	-62.	END OF TAPE DETECTED
IE.OFL	177677	-65.	DEVICE OFF LINE
IE.BCC	177676	-66.	BLOCK CHECK, CRC, OR FRAMING ERROR
IE.NOD	177751	-23.	CALLER'S NODES EXHAUSTED
IE.DFU	177750	-24.	DEVICE FULL
IE.IFU	177747	-25.	INDEX FILE FULL
IE.NSF	177746	-26.	NO SUCH FILE
IE.LCK	177745	-27.	LOCKED FROM READ/WRITE ACCESS
IE.HFU	177744	-28.	FILE HEADER FULL
IE.WAC	177743	-29.	ACCESSED FOR WRITE
IE.CKS	177742	-30.	FILE HEADER CHECKSUM FAILURE
IE.WAT	177741	-31.	ATTRIBUTE CONTROL LIST FORMAT ERROR
IE.RER	177740	-32.	FILE PROCESSOR DEVICE READ ERROR
IE.WER	177737	-33.	FILE PROCESSOR DEVICE WRITE ERROR
IE.ALN	177736	-34.	FILE ALREADY ACCESSED ON LUN
IE.SNC	177735	-35.	FILE ID, FILE NUMBER CHECK
IE.SQC	177734	-36.	FILE ID, SEQUENCE NUMBER CHECK
IE.NLN	177733	-37.	NO FILE ACCESSED ON LUN
IE.CLO	177732	-38.	FILE WAS NOT PROPERLY CLOSED
IE.DUP	177707	-57.	ENTER - DUPLICATE ENTRY IN DIRECTORY
IE.BVR	177701	-63.	BAD VERSION NUMBER
IE.BHD	177700	-64.	BAD FILE HEADER
IE.EXP	177665	-75.	FILE EXPIRATION DATE NOT REACHED
IE.BTF	177664	-76.	BAD TAPE FORMAT
IE.ALC	177654	-84.	ALLOCATION FAILURE
IE.ULK	177653	-85.	UNLOCK ERROR
IE.WCK	177652	-86.	WRITE CHECK FAILURE

IE.NBF	177731	-39.	OPEN - NO BUFFER SPACE AVAILABLE FOR FILE
IE.RBG	177730	-40.	ILLEGAL RECORD SIZE
IE.NBK	177727	-41.	FILE EXCEEDS SPACE ALLOCATED, NO BLOCKS
IE.ILL	177726	-42.	ILLEGAL OPERATION ON FILE DESCRIPTOR BLOCK
IE.BTP	177725	-43.	BAD RECORD TYPE
IE.RAC	177724	-44.	ILLEGAL RECORD ACCESS BITS SET
IE.RAT	177723	-45.	ILLEGAL RECORD ATTRIBUTES BITS SET
IE.RCN	177722	-46.	ILLEGAL RECORD NUMBER - TOO LARGE
IE.2DV	177720	-48.	RENAME - 2 DIFFERENT DEVICES
IE.FEX	177717	-49.	RENAME - NEW FILE NAME ALREADY IN USE
IE.BDR	177716	-50.	BAD DIRECTORY FILE
IE.RNM	177715	-51.	CAN'T RENAME OLD FILE SYSTEM
IE.BDI	177714	-52.	BAD DIRECTORY SYNTAX
IE.FOP	177713	-53.	FILE ALREADY OPEN
IE.BNM	177712	-54.	BAD FILE NAME
IE.BDV	177711	-55.	BAD DEVICE NAME
IE.NFI	177704	-60.	FILE ID WAS NOT SPECIFIED
IE.ISQ	177703	-61.	ILLEGAL SEQUENTIAL OPERATION
IE.NNC	177663	-77.	NOT ANSI 'D' FORMAT BYTE COUNT
IE.AST	177660	-80.	NO AST SPECIFIED IN CONNECT
IE.NNN	177674	-68.	NO SUCH NODE
IE.NFW	177673	-69.	PATH LOST TO PARTNER
IE.BLB	177672	-70.	BAD LOGICAL BUFFER
IE.TMM	177671	-71.	TOO MANY OUTSTANDING MESSAGES
IE.NDR	177670	-72.	NO DYNAMIC SPACE AVAILABLE
IE.CNR	177667	-73.	CONNECTION REJECTED
IE.TMO	177666	-74.	TIMEOUT ON REQUEST
IE.NNL	177662	-78.	NOT A NETWORK LUN
IE.NLK	177661	-79.	TASK NOT LINKED TO SPECIFIED ICS/ICR INTERRUPTS
IE.NST	177660	-80.	SPECIFIED TASK NOT INSTALLED
IE.FLN	177657	-81.	DEVICE OFFLINE WHEN OFFLINE REQUEST WAS ISSUED
IE.IES	177656	-82.	INVALID ESCAPE SEQUENCE
IE.PES	177655	-83.	PARTIAL ESCAPE SEQUENCE
IE.ICE	177721	-47.	INTERNAL CONSISTENCY ERROR
IE.ONL	177675	-67.	DEVICE ONLINE
IE.NTR	177651	-87.	TASK NOT TRIGGERED
IE.REJ	177650	-88.	TRANSFER REJECTED BY RECEIVING CPU
IE.FLG	177647	-89.	EVENT FLAG ALREADY SPECIFIED

C.4.3 Summary of RMS-11 Error Codes

RMS–11 error codes are returned during run time in offset STS of the File Access Block (FAB) or Record Access Block (RAB). Additional status information or system error codes are returned in offset STV.

ER$ABO	177760	-16.	OPERATION ABORTED (STV=ER$STK/MAP)
ER$ACC	177740	-32.	F11ACP COULD NOT ACCESS FILE (STV=SYS ERR CODE)
ER$ACT	177720	-48.	"FILE" ACTIVITY PRECLUDES OPERATION
ER$AID	177700	-64.	BAD AREA ID (STV=@XAB)
ER$ALN	177660	-80.	ALIGNMENT OPTIONS ERROR (STV=@XAB)
ER$ALQ	177640	-96.	ALLOCATION QUANTITY TOO LARGE
ER$ANI	177620	-112.	NOT ANSI "D" FORMAT
ER$AOP	177600	-128.	ALLOCATION OPTIONS ERROR (STV=@XAB)
ER$AST	177560	-144.	INVALID (I.E. SYNCH) OPERATION AT AST LEVEL
ER$ATR	177540	-160.	ATTRIBUTE READ ERROR (STV=SYS ERR CODE)
ER$ATW	177520	-176.	ATTRIBUTE WRITE ERROR (STV=SYS ERR CODE)
ER$BKS	177500	-192.	BUCKET SIZE TOO LARGE (FAB)
ER$BKZ	177460	-208.	BUCKET SIZE TOO LARGE (STV=@XAB)
ER$BLN	177440	-224.	"BLN" LENGTH ERROR (RAB/FAB)
ER$BOF	177430	-232.	BEGINNING OF FILE DETECTED($SPACE)
ER$BPA	177420	-240.	PRIVATE POOL ADDRESS NOT MULTIPLE OF "4"
ER$BPS	177400	-256.	PRIVATE POOL SIZE NOT MULTIPLE OF "4"
ER$BUG	177360	-272.	INTERNAL RMS ERROR CONDITION DETECTED
ER$CCR	177340	-288.	CAN'T CONNECT RAB
ER$CHG	177320	-304.	$UPDATE-KEY CHANGE WITHOUT HAVING ATTRIBUTE OF XB$CHG SET
ER$CHK	177300	-320.	BUCKET FORMAT CHECK-BYTE FAILURE
ER$CLS	177260	-336.	RSTS/E CLOSE FUNCTION FAILED (STV=SYS ERR CODE)
ER$COD	177240	-352.	INVALID OR UNSUPPORTED "COD" FIELD (STV=@XAB)
ER$CRE	177220	-368.	COULD NOT CREATE FILE (STV=SYS ERR CODE)
ER$CUR	177200	-384.	NO CURRENT RECORD (OPERATION NOT PRECEDED BY GET/FIND)
ER$DAC	177160	-400.	F11-ACP DEACCESS ERROR DURING "CLOSE" (STV=SYS ERR CODE)
ER$DAN	177140	-416.	DATA "AREA" NUMBER INVALID (STV=@XAB)
ER$DEL	177120	-432.	RFA-ACCESSED RECORD WAS DELETED
ER$DEV	177100	-448.	BAD DEVICE, OR INAPPROPRIATE DEVICE TYPE
ER$DFW	177070	-456.	ERROR OCCURRED ON DEFERRED WRITE (STV=SYS ERR CODE)
ER$DIR	177060	-464.	ERROR IN DIRECTORY NAME
ER$DME	177040	-480.	DYNAMIC MEMORY EXHAUSTED
ER$DNF	177020	-496.	DIRECTORY NOT FOUND
ER$DNR	177000	-512.	DEVICE NOT READY
ER$DPE	176770	-520.	DEVICE POSITIONING ERROR (STV=SYS ERR CODE)
ER$DTP	176760	-528.	"DTP" FIELD INVALID (STV=@XAB)
ER$DUP	176740	-544.	DUPLICATE KEY DETECTED, XB$DUP ATTRIBUTE NOT SET
ER$ENT	176720	-560.	RSX-F11ACP ENTER FUNCTION FAILED (STV=SYS ERR CODE)
ER$ENV	176700	-576.	OPERATION NOT SELECTED IN "ORG$" MACRO
ER$EOF	176660	-592.	END-OF-FILE
ER$ESS	176640	-608.	EXPANDED STRING AREA TOO SHORT
ER$EXP	176630	-616.	FILE EXPIRATION DATE NOT YET REACHED
ER$EXT	176620	-624.	FILE EXTEND FAILURE (STV=SYS ERR CODE)
ER$FAB	176600	-640.	NOT A VALID FAB ("BID" NOT=FB$BID)
ER$FAC	176560	-656.	ILLEGAL FAC FOR REC-OP,0, OR FB$PUT NOT SET FOR "CREATE"
ER$FEX	176540	-672.	FILE ALREADY EXISTS
ER$FID	177530	-680.	INVALID FILE-ID

ER$FLG	176520	-688.	INVALID FLAG-BITS COMBINATION (STV=@XAB)
ER$FLK	176500	-704.	FILE IS LOCKED BY OTHER USER
ER$FND	176460	-720.	RSX-F11ACP "FIND" FUNCTION FAILED (STV=SYS ERR CODE)
ER$FNF	176440	-736.	FILE NOT FOUND
ER$FNM	176420	-752.	ERROR IN FILE NAME
ER$FOP	176400	-768.	INVALID FILE OPTIONS
ER$FSS	176370	-776.	SYSTEM ERROR DURING FNA/DNA STRING PARSE (STV=SYS ERR CODE)
ER$FUL	176360	-784.	DEVICE/FILE FULL
ER$IAN	176340	-800.	INDEX "AREA" NUMBER INVALID (STV=@XAB)
ER$IDX	176320	-816.	INDEX NOT INITIALIZED (STV ONLY, STS=ER$RNF)
ER$IFI	176300	-832.	INVALID IFI VALUE, OR UNOPENED FILE
ER$IMX	176260	-848.	MAX NUM (254) AREAS/KEY XABS EXCEEDED (STV=@XAB)
ER$INI	176240	-864.	$INIT MACRO NEVER ISSUED
ER$IOP	176220	-880.	OPERATION ILLEGAL, OR INVALID FOR FILE ORG.
ER$IRC	176200	-896.	ILLEGAL RECORD ENCOUNTERED (SEQ. FILES ONLY)
ER$ISI	176160	-912.	INVALID ISI VALUE, OR UNCONNECTED RAB
ER$KBF	176140	-928.	BAD KEY BUFFER ADDRESS (KBF=0)
ER$KEY	176120	-944.	INVALID KEY FIELD (KEY=0/NEG)
ER$KRF	176100	-960.	INVALID KEY-OF-REFERENCE ($GET/$FIND)
ER$KSZ	176060	-976.	KEY SIZE=0, OR TOO LARGE (IDX)/NOT=4(REL)
ER$LAN	176040	-992.	LOWEST-LEVEL-INDEX "AREA" NUMBER INVALID (STV=@XAB)
ER$LBL	176020	-1008.	NOT ANSI LABELED TAPE
ER$LBY	176000	-1024.	LOGICAL CHANNEL BUSY
ER$LCH	175760	-1040.	LOGICAL CHANNEL NUMBER TOO LARGE
ER$LEX	175750	-1048.	LOGICAL EXTEND ERROR, PRIOR EXTEND STILL VALID (STV=@XAB)
ER$LOC	175740	-1056.	"LOC" FIELD INVALID (STV=@XAB)
ER$MAP	175720	-1072.	BUFFER MAPPING ERROR
ER$MKD	175700	-1088.	F11ACP COULD NOT MARK FILE FOR DELETION (STV=SYS ERR CODE)
ER$MRN	175660	-1104.	MRN VALUE=NEG/REL.KEY>MRN
ER$MRS	175640	-1120.	MRS VALUE=0 FOR FIXED LENGTH RECS/=0 FOR REL. FILES
ER$NAM	175620	-1136.	"NAM" BLOCK ADDRESS INVALID (NAM=0, OR NOT ACCESSIBLE)
ER$NEF	175600	-1152.	NOT POSITIONED TO EOF (SEQ. FILES ONLY)
ER$NID	175560	-1168.	CAN'T ALLOCATE INTERNAL INDEX DESCRIPTOR
ER$NPK	175540	-1184.	INDEXED FILE-NO PRIMARY KEY DEFINED
ER$OPN	175520	-1200.	RSTS/E OPEN FUNCTION FAILED (STV=SYS ERR CODE)
ER$ORD	175500	-1216.	XAB'S NOT IN CORRECT ORDER (STV=@XAB)
ER$ORG	175460	-1232.	INVALID FILE ORGANIZATION VALUE
ER$PLG	175440	-1248.	ERROR IN FILE'S PROLOGUE (RECONSTRUCT FILE)
ER$POS	175420	-1264.	"POS" FIELD INVALID (POS>MRS,STV=@XAB)
ER$PRM	175400	-1280.	BAD FILE DATE FIELD RETRIEVED (STV=@XAB)
ER$PRV	175360	-1296.	PRIVILEGE VIOLATION (OS DENYS ACCESS)
ER$RAB	175340	-1312.	NOT A VALID RAB ("BID" NOT=RB$BID)
ER$RAC	175320	-1328.	ILLEGAL RAC VALUE
ER$RAT	175300	-1344.	ILLEGAL RECORD ATTRIBUTES
ER$RBF	175260	-1360.	INVALID RECORD BUFFER ADDR (NOT WORD-ALIGNED IF BLK-IO)
ER$RER	175240	-1376.	FILE READ ERROR (STV=SYS ERR CODE)
ER$REX	175220	-1392.	RECORD ALREADY EXISTS
ER$RFA	175200	-1408.	BAD RFA VALUE (RFA=0)
ER$RFM	175160	-1424.	INVALID RECORD FORMAT
ER$RLK	175140	-1440.	TARGET BUCKET LOCKED BY ANOTHER STREAM
ER$RMV	175120	-1456.	RSX-F11ACP REMOVE FUNCTION FAILED (STV=SYS ERR CODE)
ER$RNF	175100	-1472.	RECORD NOT FOUND (STV=0/ER$IDX)
ER$RNL	175060	-1488.	RECORD NOT LOCKED
ER$ROP	175040	-1504.	INVALID RECORD OPTIONS
ER$RPL	175020	-1520.	ERROR WHILE READING PROLOGUE (STV=SYS ERR CODE)

ER$RRV	175000	-1536.	INVALID RRV RECORD ENCOUNTERED
ER$RSA	174760	-1552.	RAB STREAM CURRENTLY ACTIVE
ER$RSZ	174740	-1568.	BAD RECORD SIZE (RSZ>MRS, OR NOT=MRS IF FIXED LENGTH RECS
ER$RTB	174720	-1584.	RECORD TOO BIG FOR USER'S BUFFER (STV=ACTUAL REC SIZE)
ER$RVU	174710	-1592.	RRV UPDATE ERROR ON INSERT
ER$SEQ	174700	-1600.	PRIMARY KEY OUT OF SEQUENCE (RAC=RB$SEQ FOR $PUT)
ER$SHR	174660	-1616.	"SHR" FIELD INVALID FOR FILE (CAN'T SHARE SEQ FILES)
ER$SIZ	174640	-1632.	"SIZ" FIELD INVALID (STV=@XAB)
ER$STK	174620	-1648.	STACK TOO BIG FOR SAVE AREA
ER$SYS	174600	-1664.	SYSTEM DIRECTIVE ERROR (STV=SYS ERR CODE)
ER$TRE	174560	-1680.	INDEX TREE ERROR
ER$TYP	174540	-1696.	ERROR IN FILE TYPE EXTENSION/FNS TOO BIG
ER$UBF	174520	-1712.	INVALID USER BUFFER ADDR (0, OR BLK-IO NOT WORD ALIGNED)
ER$USZ	174500	-1728.	INVALID USER BUFFER SIZE (USZ=0)
ER$VER	174460	-1744.	ERROR IN VERSION NUMBER
ER$VOL	174440	-1760.	INVALID VOLUME NUMBER (STV=@XAB)
ER$WCD	174430	-1768.	WILD CARD ENCOUNTERED DURING FNA/DNA STRING PARSE
ER$WER	174420	-1776.	FILE WRITE ERROR (STV=SYS ERR CODE)
ER$WLK	174410	-1784.	DEVICE IS WRITE-LOCKED
ER$WPL	174400	-1792.	ERROR WHILE WRITING PROLOGUE (STV=SYS ERR CODE)
ER$XAB	174360	-1808.	NOT A VALID XAB (@XAB=ODD,STV=@XAB)
ER$XTR	174340	-1824.	EXTRANEOUS FIELD DETECTED DURING FNA/DNA STRING PARSE

Appendix D

System Subroutines

D.1 System Subroutine Summary

The FORTRAN-77 library contains, in addition to functions intrinsic to the FORTRAN language, subroutines that the user may call (except on RSTS/E) in the same manner as a user-written subroutine. These subroutines are described in this appendix.

In addition, the RSX-11 operating systems provide a complete set of subroutines, callable from FORTRAN, for performing process control and executive calls (see the *RSX-11M/M-PLUS Executive Reference Manual*).

The following subroutines are supplied with FORTRAN-77.

ASSIGN Specifies, at run time, device and/or file name information to be associated with a logical unit number.

CLOSE Closes a file on a specified logical unit.

DATE Returns a 9-byte string containing the ASCII representation of the current date.

IDATE Returns three integer values representing the current month, day, and year.

ERRSET Specifies the action to be taken on detection of certain errors.

ERRSNS Returns information about the most recently detected error condition.

ERRTST Returns information about whether a specific error condition has occurred during program execution.

EXIT Terminates the execution of a program, reports termination status information, and returns control to the operating system.

USEREX Specifies a user subprogram to be called immediately prior to task termination.

FDBSET Specifies special I/O options to be associated with a logical unit.

RAD50 Converts 6-character Hollerith strings to Radix-50 representation and returns the result as a function value.

IRAD50 Converts Hollerith strings to Radix–50 representation.

R50ASC Converts Radix–50 strings to Hollerith strings.

SECNDS Provides system time of day or elapsed time as a floating-point function value, in seconds.

TIME Returns an 8-byte string containing the ASCII representation of the current time, in hours, minutes, and seconds.

References to integer arguments in the following subroutine descriptions refer to arguments of type INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines, if their value is within the INTEGER*2 range. However, arguments that receive return values from these subroutines must, for correct operation, be INTEGER*2 variables or array elements.

D.2 ASSIGN

The ASSIGN subroutine specifies file name information for a logical unit. The ASSIGN call must be executed before the logical unit is opened for I/O operations. The assignment remains in effect until the end of the program or until the file is closed by the CLOSE subroutine or a CLOSE statement. The call to ASSIGN has the form:

```
CALL ASSIGN(n[,name][,icnt])
```

n
An integer value that specifies the logical unit a number.

name
A variable, array, array element, or alphanumeric literal that contains any standard file specification. If the device is not specified, the device assignment remains unchanged. If a file name is not specified, the default name as described in Section 2.1.1 is used.

icnt
An INTEGER*2 value that specifies the number of characters in the string name. If icnt is zero or not present, the string name is processed until the first ASCII null character is encountered.

CALL ASSIGN requires only the first argument; all others are optional and, if omitted, are replaced by the default values as noted in the argument descriptions. However, if any argument is to be included, all arguments that precede it must also be included.

If only the unit number argument is specified, all previously specified file name information concerning that unit is disassociated from the unit, and the default conditions become effective.

For example, in the following situation:

```
CALL ASSIGN(5,'SY:ABC.DAT')
WRITE(5,-) ....
CALL CLOSE(5)
WRITE(5,-) ....
```

the first WRITE operation is performed to file ABC.DAT and the second to FOR005.DAT.

See also the discussion in Section 2.1.1 concerning default device assignments.

D.3 CLOSE

The CLOSE subroutine closes the currently open file on a logical unit. The call to CLOSE has the form:

```
CALL CLOSE(n)
```

n
An integer value that specifies the logical unit number.

When the close is completed, the logical unit reacquires the default file name attributes in effect when program execution was initiated.

See also the discussion in Section 2.1.1 concerning default device assignments.

D.4 DATE

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form:

```
CALL DATE(buf)
```

buf
An array or array element.

The date is returned as a 9-byte ASCII string of the form:

```
dd-mmm-yy
```

dd
The 2-digit date.

mmm
The 3-letter month specification.

yy
The last two digits of the year.

D.5 IDATE

The IDATE subroutine returns three INTEGER*2 values that represent the current month, day, and year. The call to IDATE has the form:

```
CALL IDATE(i,j,k)
```

If the current date is October 19, 1988, the values of the integer variables upon return are:

```
i = 3
j = 19
k = 79
```

D.6 ERRSET

The ERRSET subroutine specifies the action to be taken when an error is detected by the OTS. The error action to be taken is specified individually for each error–that is, independently of other errors. The call to ERRSET has the form:

```
CALL ERRSET(number, contin, count, type, log, maxlim)
```

number
An integer value that specifies the error number to which the following parameters apply.

contain
A logical value that specifies whether to continue after an error. .TRUE. means continue after the error is detected; .FALSE. causes an exit after the error.

count
A logical value that specifies whether to count this error against the task's maximum error limit. .TRUE. causes the error to be counted; .FALSE. causes it not to be counted.

type
A logical value that specifies the type of continuation to be performed after error detection. .TRUE. passes control to an ERR= transfer label if available; .FALSE. causes a return to the routine that detected the error for default error recovery.

log
A logical value that specifies whether to produce an error message for this error. .TRUE. produces a message; .FALSE. suppresses the message.

maxlim
A positive INTEGER*2 value used to set the task's maximum error limit. The default value is set at 15 at task initialization.

Null arguments are permitted for all but the first argument and cause no change in the current state of that control code.

See Section 3.5 for a discussion of the control effects obtained by these subroutine arguments. Table 3–2 shows the initial settings of the error control bits.

D.7 ERRSNS

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form:

```
CALL ERRSNS (num,ioerr,ioerr1,iunit)
```

num
An INTEGER*2 variable or array element name in which the most recent error number is stored. A zero will be returned if no error has occurred since the last call to ERRSNS, or if no error has occurred since the beginning of task execution.

If the last error occurred as a result of an I/O error, the next three parameters receive selected values. Otherwise, values of 0 are returned.

ioerr
An INTEGER*2 variable or array element in which the primary file system error code is stored: that is, the FCS-11 F.ERR value or the RMS-11 STS value.

ioerr
An INTEGER*2 variable or array element in which the secondary file system error code is stored: that is, the FCS-11 F.ERR+1 value or the RMS–11 STV value.

iunit
An INTEGER*2 variable or array element in which the logical unit number is stored.

From zero to four arguments may be specified. After the call to ERRSNS, the error information is reset to 0.

To determine if an error occurs in a given section of a program, the following technique is suggested:

1. Call ERRSNS immediately prior to the segment in order to clear any previous error data.
2. Execute the section.
3. Call ERRSNS again and branch on a nonzero return value to error analysis code.

For example:

```
CALL ERRSNS
CALL ASSIGN (1,'NAME.DAT')
CALL FDBSET (1,'OLD','SHARE')
CALL ERRSNS (IERR,IFCS,IFCS1,ILUN)
IF (IERR.NE.0) GOTO 100
```

D.8 ERRTST

The ERRTST subroutine tests for the occurrence of a specific error during program execution. The call to ERRTST has the form:

```
CALL ERRTST(i,j)
```

i
The INTEGER*2 error number, and the value of j is returned as:

1 if error number i has occurred
2 if error number i has not occurred

For example, the sequence

```
          .
          .
          .
     CALL ERRTST(43,J)
     GO TO (10,20),J
20   CONTINUE
          .
          .
          .
```

transfers control to statement 10 if error 43 has occurred.

The ERRTST routine also resets to 0 the error flag for an occurring error. For example, in the sequence

```
          .
          .
          .
     CALL ERRTST(I,J)
     CALL ERRTST(I,J)
          .
          .
          .
```

the second call is guaranteed to return J=2. The ERRTST subroutine is independent of the ERRSET subroutine; neither subroutine directly influences the other except that ERRSET can cause execution to terminate.

D.9 EXIT

The EXIT subroutine causes program termination, closes all files, reports termination status to the operating system, and returns control to the operating system. The call to EXIT has the form:

```
CALL EXIT [(istat)]
```

istat
An INTEGER*2 value that is the termination status value to be reported to the operating system.

If istat is not specified, the termination status value is success.

D.10 USEREX

The USEREX subroutine specifies a routine that is to be called as part of the program termination process. Using USEREX allows clean-up operations in non-FORTRAN routines. The call to USEREX has the form:

```
EXTERNAL name
CALL USEREX (name)
```

name
The routine that is to be called. This name must appear in an EXTERNAL statement in the program unit.

The user exit subroutine is called with a JSR PC instruction after all procedures required for FORTRAN program termination have been completed–that is, when all files have been closed, and any attempt to perform FORTRAN I/O operations produces unpredictable results. In addition, all OTS error handling is disabled; so if an error occurs in the USEREX-specified routine, the task is immediately aborted by the operating system. The transfer of control takes place immediately preceding the exit to the operating system; return from the subroutine by an RTS PC results in a normal exit to the operating system.

D.11 FDBSET

The FDBSET subroutine specifies special input/output options. (It is provided primarily for compatibility with older FORTRAN implementations because similar and more extensive capabilities are available through the OPEN statement.) The call to FDBSET has the form:

```
CALL FDBSET(unit,mode,share,numbuf,initsz,extend)
```

unit
an INTEGER*2 value specifying the logical unit to which the subsequent arguments apply.

mode
One of the following character constants, specifying the type of access to be used:

```
'READONLY'   For read-only access.
'NEW'        For creating a new file.
'OLD'        For accessing an existing file.
'APPEND'     For appending to an existing sequential file.
'UNKNOWN'    For an unknown file; has the effect of trying
             'OLD' first, and if no such file exists, uses 'NEW'.
```

share
The character constant 'SHARE', which specifies that shared access is allowed.

numbuf
An INTEGER*2 value that specifies the number of buffers to be used for multibuffered input/output.

initsz
An INTEGER*2 value that specifies the initial allocation, in disk blocks, of file storage for a new file.

extend
An INTEGER*2 value that specifies the number of blocks by which to extend a file.

FDBSET may only be called prior to opening the unit specified in the first argument. CALL FDBSET, CALL ASSIGN, and the DEFINEFILE statement may be used together.

The unit number argument is required. All other arguments may be null or missing to indicate no specification for that argument.

D.12 IRAD50

The IRAD50 subprogram performs conversions of ASCII data to Radix–50 representation. Radix–50 representation is required by the Process Control subroutines and the System Directives for specifying task names within the RSX–11 system. (See Section A.5.)

IRAD50 may be called as a FUNCTION subprogram if the return value is desired, or as a SUBROUTINE subprogram if no return value is desired. The call to IRAD50 has the form:

```
n = IRAD50 (icnt,input,output)
```

or

```
    CALL IRAD50(icnt,input,output)
```

icnt
The INTEGER*2 maximum number of characters to convert.

input
An ASCII (Hollerith) text string to be converted to Radix–50.

output
The location for storing the results of the conversion.

n
The INTEGER*2 number of characters actually converted.

Three characters of text are packed into each word of output. The number of output words modified is computed by the expression (in integer mode)

```
    (icnt+2)/3
```

Therefore, if a count of four is specified, two words of output are written even if only a 1-character input string is given as an argument.

Scanning of input characters terminates on the first non-Radix–50 character encountered in the input string.

D.13 RAD50

The RAD50 function subprogram provides a simplified way to encode RSX–11 task names in Radix–50 notation (see Section A.5). This function converts six characters of ASCII data to two words of Radix–50 data. The call to RAD50 has the form:

```
    RAD50(name)
```

name
The variable name or array element corresponding to an ASCII string.

Note that the RAD50 function may be used as an argument to an RSX–11 system directive subroutine. For example:

```
    REAL*8 A
    DATA A/'TASK A'/
    CALL REQUES (RAD50(A),...)
```

The RAD50 function is equivalent to the following FORTRAN function:

```
FUNCTION RAD50(A)
CALL IRAD50(6,A,RAD50)
RETURN
END
```

D.14 R50ASC

The R50ASC subprogram provides decoding of Radix-50 encoded values into ASCII strings. The call to R50ASC has the form:

```
CALL R50ASC (icnt,in,out)
```

icnt
The INTEGER*2 number of output characters to be produced.

in
The variable or array that contains the encoded input. Note that (icnt+2)/3 words are read for conversion.

out
The variable or array in which icnt characters (bytes) are placed.

If the undefined Radix-50 code is detected, or the Radix-50 word exceeds maximum value 174777 (octal), question marks are placed in the output.

D.15 SECNDS

The SECNDS function subprogram returns the system time in seconds as a single-precision, floating-point value less the value of its single-precision, floating-point argument. The call to SECNDS has the form:

```
REAL SECNDS
y = SECNDS(x)
```

y
Set equal to the time in seconds since midnight, minus the user-supplied value of x.

You can use the SECNDS function to perform elapsed-time computations. For example:

```
C      START OF TIMED SEQUENCE
       T1 = SECNDS(0.0)

C
C      CODE TO BE TIMED
C
       DELTA = SECNDS(T1)
```

where DELTA gives the elapsed time.

The value of SECNDS is accurate to the resolution of the system clock: 0.0166 . . . seconds for a 60-cycle clock, 0.02 seconds for a 50-cycle clock.

NOTE

> The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.
>
> The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

D.16 TIME

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form:

```
CALL TIME(buf)
```

buf
An 8-byte variable, array, or array element.

The TIME call returns the time as an 8-byte ASCII character string of the form:

```
hh:mm:ss
```

hh
The 2-digit hour indication.

mm
The 2-digit minute indication.

ss
The 2-digit second indication.

For example:

```
10:45:23
```

A 24-hour clock is used.

Appendix E

Compatibility: PDP–11 FORTRAN–77 and PDP–11 FORTRAN IV-PLUS

PDP–11 FORTRAN–77 is based on American National Standard FORTRAN–77, X3.9-1978. As a result, it contains certain incompatibilities with the PDP–11 FORTRAN IV-PLUS language, which is based on the previous standard, X3.9-1966. The areas affected are:

- DO loop minimurr. iteration count
- EXTERNAL statement
- OPEN statement BLANK keyword default
- OPEN statement STATUS keyword default
- Blank common block PSECT
- X format edit descriptor

The PDP–11 FORTRAN–77 compiler selects ANSI FORTRAN–77 language interpretations by default. If you are compiling PDP–11 FORTRAN IV-PLUS programs, there are several actions you can take to compensate for language incompatibilities:

- You can modify your programs so that they produce the intended result with the /F77 switch. Compiler diagnostics help you identify OPEN statements in which an explicit STATUS keyword should be added. Linker diagnostics help you locate EXTERNAL statements that must be changed to INTRINSIC statements.
- You can specify the /NOF77 switch to select PDP–11 FORTRAN IV-PLUS language interpretations. The /NOF77 switch affects the interpretation of DO loop minimum iteration counts, EXTERNAL statements, and OPEN statement BLANK and STATUS defaults. It does not affect the X format edit descriptor.

- You can build the PDP–11 FORTRAN–77 compiler with the /NOF77 switch as the default, which selects PDP–11 FORTRAN IV-PLUS language interpretations as defaults.

This appendix discusses each of the language differences. When possible, it gives an example of how you can modify your PDP–11 FORTRAN IV-PLUS programs to make them compatible with both PDP–11 FORTRAN–77 and PDP–11 FORTRAN IV-PLUS.

E.1 DO Loop Minimum Iteration Count

In PDP–11 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 4.4.2). In PDP–11 FORTRAN IV-PLUS, however, the body of a DO loop is always executed at least once.

If you are running a PDP–11 FORTRAN IV-PLUS program with the /F77 switch, you may want to ensure a minimum loop count of one by modifying the program's DO statements. As an example, assume that a FORTRAN IV-PLUS program contains this statement:

```
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 PDP–11 FORTRAN–77 and PDP–11 FORTRAN IV-PLUS.

The /F77 switch controls the interpretation of the DO loop minimum iteration count.

E.2 EXTERNAL Statement

Under PDP–11 FORTRAN IV-PLUS, a function specified in an EXTERNAL statement with the name of a FORTRAN processor-defined (intrinsic) or library function was assumed to refer to the named processor-defined or library function, not to a user-defined function with that name. If, however, a function name appeared in an EXTERNAL statement preceded by an asterisk, that function was assumed to be a user-defined function, regardless of any name conflicts.

Under ANSI FORTRAN–77 and PDP–11 FORTRAN–77, a function specified in an EXTERNAL statement with the name of a processor-defined (intrinsic) or library function is assumed to refer to a user-defined function.

Under PDP–11 FORTRAN–77, the function name fname in the statement

```
EXTERNAL fname [,fname ...]
```

is interpreted to refer to a user-defined function by default.

If the /NOF77 switch is specified, and fname is the same as one of the processor-defined or library functions, fname is interpreted to refer to the processor-defined or library function.

If fname appears preceded by an asterisk, it is interpreted to refer to a user-defined function if the /NOF77 switch is set, but it is an error if the /F77 switch is set.

All functions declared with the new INTRINSIC statement are interpreted to be processor-defined (intrinsic) or library functions, regardless of the setting of the/NOF77 switch.

E.3 OPEN Statement BLANK Keyword Default

In PDP–11 FORTRAN–77, the OPEN statement BLANK keyword controls the interpretation of blanks in numeric input fields. The PDP–11 FORTRAN–77 default is BLANK='NULL'; that is, blanks in numeric input fields are ignored. The PDP–11 FORTRAN IV-PLUS OPEN statement does not have a BLANK keyword. However, the PDP–11 FORTRAN IV-PLUS interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

If a logical unit is opened without an explicit OPEN statement, PDP-11 FORTRAN-77 and PDP-11 FORTRAN IV-PLUS 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 PDP-11 FORTRAN-77 and in PDP-11 FORTRAN IV-PLUS:

Program:

```
        OPEN(UNIT=1, STATUS='OLD') READ(1,10)I, J
     10 FORMAT (215) END
```

Data record:

```
      1 2   12
```

FORTRAN-77 Values	FORTRAN IV-PLUS Values
I = 12	I = 1020
J = 12	J = 12

The /F77 switch controls 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 the /NOF77 switch, include BLANK='ZERO' in the OPEN statement or use the BZ edit descriptor in the FORMAT statement.

E.4 OPEN Statement STATUS Keyword Default

In PDP-11 FORTRAN-77, the OPEN statement STATUS keyword specifies the initial status of the file ('OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'). The PDP-11 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. The PDP-11 FORTRAN IV-PLUS keyword TYPE is a synonym for STATUS; however, the PDP-11 FORTRAN IV-PLUS default is TYPE='NEW'.

If you use the /F77 switch 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.

The /F77 switch controls the default value for the STATUS (or TYPE) keyword.

E.5 Blank Common Block PSECT (.$$$$.)

Under PDP–11 FORTRAN–77, the blank common block PSECT (.$$$$.) has the SAV attribute; it does not have this attribute under PDP–11 FORTRAN IV-PLUS. The SAV attribute on a PSECT has the effect of pulling that PSECT into the root segment of an overlay.

The /F77 command switch controls the default assignment of the SAV attribute; under /F77, the blank common block PSECT is assigned the SAV attribute by default.

E.6 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 PDP–11 FORTRAN–77 output statement, character positions that are skipped are not modified, and the length of the output record is not affected. However, in a PDP–11 FORTRAN IV-PLUS output statement, the X edit descriptor writes blanks and may extend the output record. For example, the statements

```
   WRITE(1,10)
10 FORMAT(1X, 'ABCDEF', T4, 2X, '12345', 3X)
```

produce the output records:

FORTRAN-77	FORTRAN IV-PLUS
ABCD12345	AB 12345

The /F77 switch does not affect the interpretation of the X edit descriptor. To achieve the PDP–11 FORTRAN IV-PLUS effect, change nX to n(' ').

Appendix F

Compatibility: PDP-11 FORTRAN-77, PDP-11 FORTRAN IV, VAX FORTRAN

PDP-11 FORTRAN-77 is a compatible superset of PDP-11 FORTRAN IV and a compatible subset of VAX FORTRAN.

Generally speaking, any PDP-11 FORTRAN-77 program that does not use superset features runs correctly in PDP-11 FORTRAN IV, and any PDP-11 FORTRAN-77 program runs correctly in VAX FORTRAN.

Differences in execution, however, may be encountered because of differences in compiler architecture, hardware architecture, or operating system environment.

The following sections discuss differences among PDP-11 FORTRAN IV, PDP-11 FORTRAN-77, and VAX FORTRAN.

There are both language differences and run-time support differences among PDP-11 FORTRAN IV, PDP-11 FORTRAN-77, and VAX FORTRAN.

F.1 Language Differences

Differences related to language involve:

- Logical tests
- Floating-point results
- Logical unit numbers
- Assigned GO TO label list
- Integer computations
- Effect of DISPOSE = 'PRINT' specification

F.1.1 Logical Tests

The logical constants .TRUE. and .FALSE. are defined, respectively, as all 1s and all zeros by both VAX FORTRAN and PDP-11 FORTRAN. The test of .TRUE. and .FALSE. differs, however.

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: All zeros is .FALSE.; any nonzero pattern is .TRUE.. And VAX FORTRAN tests the low-order bit (bit 0) of a logical value. (This is the system-wide VAX convention for testing logical values.)

In most cases, these differences have no effect on compatibility. They are significant only for nonstandard FORTRAN programs that perform arithmetic operations on logical values and then make logical tests on the result.

In the example:

```
LOGICAL*1 BA
BA = 3
IF (BA) GO TO 10
```

PDP-11 FORTRAN-77 produces a value of .FALSE., but PDP-11 FORTRAN IV and VAX FORTRAN produce a value of .TRUE.

F.1.2 Floating-Point Results

Differences in math library routine results may occur between different arithmetic hardware configurations on PDP–11 processors and between PDP–11 and VAX hardware due to the hardware architecture differences. Equivalent accuracy is provided but there may be differences in the least-significant digits.

F.1.3 Logical Unit Numbers

If you specify a logical unit number in an I/O statement, a default unit number is used. The defaults used by PDP–11 FORTRAN–77 and PDP–11 FORTRAN IV differ from those used by VAX FORTRAN, as shown in Table F–1.

Table F–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 uses normal logical unit numbers, but VAX FORTRAN uses unit numbers that are not available to users.

F.1.4 Assigned GO TO Label List

PDP–11 FORTRAN–77 checks at run time that the label is in the list of labels specified. If not, execution continues at the next statement.

PDP–11 FORTRAN IV and VAX FORTRAN check only that the labels specified in the list are valid statement labels in the program unit. No check is made at run time, and execution continues at the label specified.

F.1.5 DISPOSE = 'Print' Specification

On some PDP-11 systems, the file is deleted after being printed if DISPOSE = 'PRINT' was specified. On VAX systems and some PDP-11 systems, the file is retained after being printed.

F.1.6 Integer Computations

In PDP-11 FORTRAN-77 and VAX FORTRAN, INTEGER*4 computations are carried out using 32-bit arithmetic. In PDP-11 FORTRAN IV, INTEGER*4 data occupies 32 bits of storage (4 bytes) but only 16 bits are used for computation.

F.1.7 Default Record Buffer Size

In PDP-11 FORTRAN-77, if there was no RECL specification when a file was created, the FORTRAN-77 OTS uses the default record size (see Section 2.3.8) as the size of the user record buffer. FORTRAN IV, however, allows the user record buffer to be as large as the value specified in the MAXBUF option in the task-build command line.

In FORTRAN-77, when you attempt to write more bytes to a record than the default record size, you should use an explicit OPEN statement with a RECL specification.

F.2 Run-Time Support Differences

Run-time support differences involve unformatted data transfer and error handling and reporting.

F.2.1 Unformatted Data Transfer

For unformatted input/output operations, four bytes of data are transferred for INTEGER*4 and LOGICAL*4 data. However, because the high-order part is undefined in PDP-11 FORTRAN IV, INTEGER*4 and LOGICAL*4 values written by a PDP-11 FORTRAN IV program may not reliably be read by PDP-11 FORTRAN-77 or VAX FORTRAN.

F.2.2 Error Handling and Reporting

Error handling and reporting differ significantly between PDP-11 FORTRAN and VAX FORTRAN. In PDP-11 FORTRAN, program execution normally continues after errors such as floating overflow until 15 such errors have occurred, at which point execution is terminated. VAX FORTRAN, however, sets a limit of one such error; program execution normally terminates when the first such error occurs.

VAX FORTRAN neither generates an error message nor increments the image error count when an I/O error occurs, if an ERR=specification is included in the I/O statement. PDP-11 FORTRAN both reports the error and increments the task error count.

Appendix G

PDP–11 FORTRAN–77 Extensions to ANSI Standard (X3.9-1978) FORTRAN

The following are PDP–11 FORTRAN–77 extensions to ANSI standard (X3.9-1978) FORTRAN at the full-language level.

If you specify the /ST switch at compile time, the compiler flags these extensions in your source code and produces informational diagnostics about them. See Section 1.2.4 for complete information on how to use the /ST switch. See Appendix C for a list of compiler diagnostic messages.

G.1 Statement Extensions

The following statements appear in PDP–11 FORTRAN–77 but not in ANSI standard FORTRAN:

ACCEPT
BYTE
DECODE
DEFINE FILE
DELETE
ENCODE
FIND
INCLUDE
REWRITE
TYPE
UNLOCK
VIRTUAL

G.2 Statement Syntax Extensions

The following sections contain PDP–11 FORTRAN–77 syntactic variations of statements present in ANSI standard FORTRAN.

G.2.1 Specification Statement

Data type *len (Except CHARACTER *len)

IMPLICIT — Examples of extended syntax follow

```
IMPLICIT INTEGER A,B
IMPLICIT INTEGER (A-C),(P-T)
```

PARAMETER — (Alternative syntax, see Section A.4, *PDP-11 FORTRAN-77 Language Reference Manual*)

typ FUNCTION nam *len(Length specifier in function declaration)

G.2.2 Format Statements

Default formats for I, F, E, D, G, L, O, A, Z

Ow, Ow.m, Q, Zw, Zw.m, $ format descriptors

P without scale factor

Variable format expressions

G.2.3 Control Statements

Null actual argument (Examples follow)

```
CALL name (,arg2)
CALL name (arg1,,arg3)
CALL name (arg1,)
CALL name (arg1,,,,arg5)
```

G.2.4 I/O Statements

READ and WRITE (Comma between I/O control and element lists; example follows)

```
READ (...), iolist
```

G.2.5 Miscellaneous Syntax Extensions

The following are present in PDP-11 FORTRAN-77 but not in ANSI standard FORTRAN:

Consecutive operators in expressions
D-line comments
End-of-line comments
Parameter constants for the real or imaginary part of a complex constant
Tab-character formatting

G.3 Keyword and Keyword Value Extensions

G.3.1 OPEN Statement Keyword Extensions

ASSOCIATEVARIABLE
BLOCKSIZE
BUFFERCOUNT
CARRIAGECONTROL
DISPOSE
DISP
EXTENDSIZE
INITIALSIZE
KEY
MAXREC
NAME
NOSPANBLOCKS
ORGANIZATION
READONLY
RECORDSIZE
RECORDTYPE
SHARED
TYPE
USEROPEN

G.3.2 OPEN Statement Keyword Value Extensions

ACCESS = 'APPEND'
ACCESS = 'KEYED'

G.3.3 CLOSE Statement Keyword Extensions

DISP
DISPOSE

G.3.4 CLOSE Statement Keyword Value Extensions

STATUS =	'SAVE'
STATUS =	'PRINT'

G.3.5 READ Statement Keyword Extensions

KEY
KEYEQ
KEYID
KEYGE
KEYGT

G.4 Lexical Extensions

The following lexical elements are present in PDP-11 FORTRAN-77 but not in ANSI standard FORTRAN:

Hollerith constants
Lowercase source letters
"nn octal constants
O octal constants
'oct'O, 'hex'X constants
Radix-50 constants
'rec in direct access I/O statements
.XOR. operator
Z hexadecimal constants

Appendix H

Software Performance Reports

From time to time, you may encounter problems and/or errors in using the FORTRAN-77 Compiler or Object Time System. These should be communicated to Digital Equipment Corporation by means of a Software Performance Report (SPR). Software Performance Report forms like the one shown in Figure H-1 may be obtained from the nearest SPR center.

You should submit Software Performance Reports to the nearest SPR center for handling. SPRs are forwarded to the appropriate group within the Software Engineering Department for analysis and response.

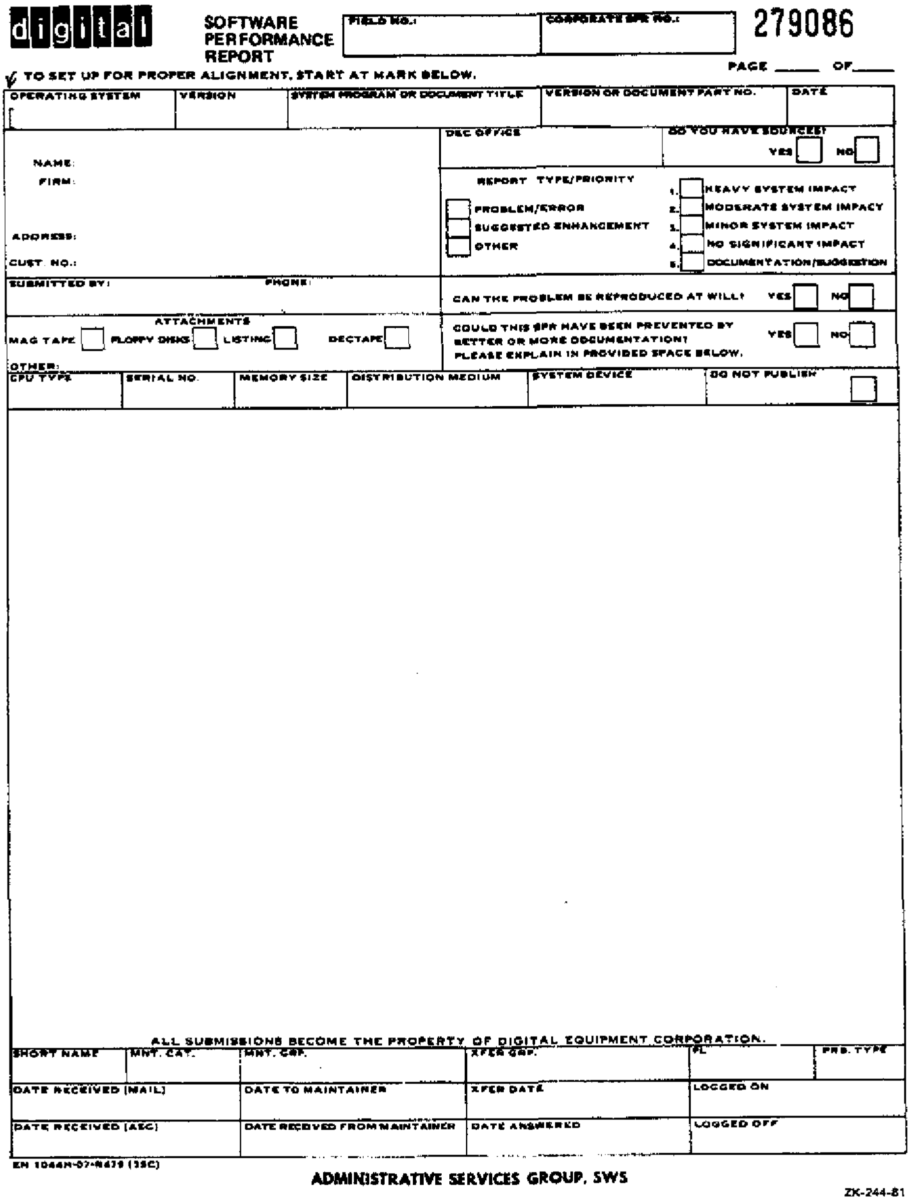
Use the following guidelines in preparing a Software Performance Report:

- Give as complete a description as possible of the problem encountered. Often a detail that may seem irrelevant will give a clue to solving the problem.
- If possible, isolate the problem to a small example. Large, unfamiliar programs are difficult to work with and may result in a misunderstanding of what the problem is or an inability to duplicate the problem.
- If the error example is longer than one page of source code, try to send all information in a machine-readable form. Machine-readable problems are much easier to diagnose and enable us to provide better service. All media are returned.
- Send console samples, command files, listings, link maps, and so on with the SPR. Annotations showing where the error occurred are extremely helpful.
- If a program reads input data, include sample input listings and, if possible, sample output.

- If an error example cannot be isolated to a single program unit, include listings for all program units involved.

Many SPRs do not contain sufficent information to duplicate or identify the problem. Complete and concise information helps DIGITAL give accurate and timely service to software problems.

Figure H–1: Software Performance Report (SPR) Form



digital

SOFTWARE
PERFORMANCE
REPORT

FIELD NO.:	CORPORATE SPR NO.:

279086

PAGE _____ OF_____

√ TO SET UP FOR PROPER ALIGNMENT, START AT MARK BELOW.

OPERATING SYSTEM	VERSION	SYSTEM PROGRAM OR DOCUMENT TITLE	VERSION OR DOCUMENT PART NO.	DATE

NAME:

FIRM:

ADDRESS:

CUST. NO.:

DEC OFFICE

DO YOU HAVE SOURCES? YES □ NO □

REPORT TYPE/PRIORITY

□ PROBLEM/ERROR

□ SUGGESTED ENHANCEMENT

□ OTHER

1. □ HEAVY SYSTEM IMPACT
2. □ MODERATE SYSTEM IMPACT
3. □ MINOR SYSTEM IMPACT
4. □ NO SIGNIFICANT IMPACT
5. □ DOCUMENTATION/SUGGESTION

SUBMITTED BY: PHONE:

CAN THE PROBLEM BE REPRODUCED AT WILL? YES □ NO □

ATTACHMENTS

MAG TAPE □ FLOPPY DISKS □ LISTING □ DECTAPE □

OTHER:

COULD THIS SPR HAVE BEEN PREVENTED BY BETTER OR MORE DOCUMENTATION? YES □ NO □
PLEASE EXPLAIN IN PROVIDED SPACE BELOW.

CPU TYPE	SERIAL NO.	MEMORY SIZE	DISTRIBUTION MEDIUM	SYSTEM DEVICE	DO NOT PUBLISH □

ALL SUBMISSIONS BECOME THE PROPERTY OF DIGITAL EQUIPMENT CORPORATION.

SHORT NAME	MNT. CAT.	MNT. GRP.	XFER GRP.	PL	PRB. TYPE
DATE RECEIVED (MAIL)		DATE TO MAINTAINER	XFER DATE	LOGGED ON	
DATE RECEIVED (ASC)		DATE RECEIVED FROM MAINTAINER	DATE ANSWERED	LOGGED OFF	

EN 1044H-07-N479 (35C)

ADMINISTRATIVE SERVICES GROUP, SWS

ZK-244-81

Index