RSX–11M–PLUS and Micro/RSX Error Logging Manual

Order No. AA–JS19A–TC

RSX–11M–PLUS Version 4.0
Micro/RSX Version 4.0

Digital Equipment Corporation Maynard, Massachusetts

First printing, January 1982
Revised, April 1983
Revised, September 1987

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 form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	EduSystem	UNIBUS
DEC/CMS	IAS	VAX
DEC/MMS	MASSBUS	VAXcluster
DECnet	MicroPDP-11	VMS
DECsystem-10	Micro/RSX	VT
DECSYSTEM-20	PDP	
DECUS	PDT	
DECwriter	RSTS	digital
DIBOL	RSX	

ZK3076

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

Chapter 3 Report Generator Task (RPT)

Chapter 4 Error Log Control File Architecture

Chapter 5 Control File Language Guide

Examples

Figures

Tables

Preface

Manual Objectives

This manual contains information about operating the RSX–11M–PLUS and Micro/RSX error logging system. It explains how the Error Logger collects information on system events and errors and how the Report Generator and control file produce various kinds of reports on those events and errors. It also includes information on the control file architecture and on how to add user-written modules. The error logging system allows you to monitor the reliability of the hardware on your system by setting error limits and displaying messages on the console terminal if the number of errors on a device exceeds those limits.

Intended Audience

This manual is intended for Field Service personnel, system managers, and others responsible for maintaining the integrity of hardware devices connected to an RSX–11M–PLUS or Micro/RSX operating system.

In addition to understanding the RSX–11M–PLUS or Micro/RSX operating system and the error logging system, you need a thorough knowledge of the hardware devices that the error logging system is monitoring. This manual does not attempt to describe or explain the hardware information that appears in the error log reports. For information about a specific device, consult the hardware documentation for that device.

Structure of This Document

Chapter 1 provides an overview of the purpose and function of the error logging system. It describes some features and limitations of the system and explains the operating system resources that error logging requires.

Chapter 2 describes the procedures for operating the Error Logger and explains the Error Log Interface (ELI) commands to control logging and limiting.

Chapter 3 describes the procedures for operating the Report Generator task (RPT) and describes the report formatting that is available.

Chapter 4 explains the control file modules in detail, including the flow of program control, interfaces between modules, and module dispatching. A knowledgeable system programmer can use the information presented to add user-written modules to the error logging system. The chapter includes extensively annotated examples of DIGITAL-supplied modules.

Chapter 5 describes the Control File Language (CFL), which is used to write control file modules.

Appendix A describes the indirect command file, TUNE.CMD, that you can use to remove devices from the error logging universal library and make it smaller.

Appendix B describes the formats used for drive serial numbers on DIGITAL devices.

Appendix C describes the formats for standard error log subpackets.

Associated Documents

This manual assumes you are familiar with the following documents:

- The *RSX-11M-PLUS Command Language Manual*
- The *Micro/RSX User's Guide*

The *RSX-11M-PLUS Information Directory and Master Index* defines the intended readership for each manual in the documentation set and provides a synopsis of each manual's contents. When this manual refers to other documents, consult the appropriate information directory for information about the document.

Conventions Used in This Document

The following conventions are used in this manual:

Convention	Meaning
>	A right angle bracket is the default prompt for the Monitor Console Routine (MCR), which is one of the command interfaces used on RSX-11M-PLUS systems. All systems include MCR.
$	A dollar sign followed by a space is the default prompt of the DIGITAL Command Language (DCL), which is one of the command interfaces used on RSX-11M-PLUS and Micro/RSX systems. Many systems include DCL.
xxx>	Three characters followed by a right angle bracket indicate the explicit prompt for a task, utility, or program on the system.
UPPERCASE	Uppercase letters in a command line indicate letters that must be entered as they are shown. For example, utility switches must always be entered as they are shown in format specifications.

Convention	Meaning
command abbreviations	Where short forms of commands are allowed, the shortest form acceptable is represented by uppercase letters. The following example shows the minimum abbreviation allowed for the DCL command DIRECTORY: `$ DIR`
lowercase	Any command in lowercase must be substituted for. Usually the lowercase word identifies the kind of substitution expected, such as a filespec, which indicates that you should fill in a file specification. For example: `filename.filetype;version` This command indicates the values that make up a file specification; values are substituted for each of these variables as appropriate.
/keyword, /qualifier, or /switch	A command element preceded by a slash (/) is an MCR keyword; a DCL qualifier; or a task, utility, or program switch. Keywords, qualifiers, and switches alter the action of the command they follow.
parameter	Required command fields are generally called parameters. The most common parameters are file specifications.
[option]	Square brackets indicate optional entries in a command line or a file specification. If the brackets include syntactical elements, such as periods (.) or slashes (/), those elements are required for the field. If the field appears in lowercase, you are to substitute a valid command element if you include the field. Note that when an option is entered, the brackets are not included in the command line.
[,...]	Square brackets around a comma and an ellipsis mark indicate that you can use a series of optional elements separated by commas. For example, (argument,[,...]) means that you can specify a series of optional arguments by enclosing the arguments in parentheses and by separating them with commas.
{ }	Braces indicate a choice of required options. You are to choose from one of the options listed.
:argument	Some parameters and qualifiers can be altered by the inclusion of arguments preceded by a colon. An argument can be either numerical (COPIES:3) or alphabetical (NAME:QIX). In DCL, the equal sign (=) can be substituted for the colon to introduce arguments. COPIES=3 and COPIES:3 are the same.

Convention	Meaning
( )	Parentheses are used to enclose more than one argument in a command line. `SET PROT = (S:RWED,O:RWED)`
,	Commas are used as separators for command line parameters and to indicate positional entries on a command line. Positional entries are those elements that must be in a certain place in the command line. Although you might omit elements that come before the desired element, the commas that separate them must still be included.
[g,m] [directory]	The convention [g,m] signifies a User Identification Code (UIC). The g is a group number and the m is a member number. The UIC identifies a user and is used mainly for controlling access to files and privileged system functions. This may also signify a User File Directory (UFD), commonly called a directory. A directory is the location of files. Other notations for directories are: [ggg,m], [gggmmm], [ufd], [name], and [directory]. The convention [directory] signifies a directory. Most directories have 1- to 9-character names, but some are in the same [g,m] form as the UIC. Where a UIC, UFD, or directory is required, only one set of brackets is shown (for example, [g,m]). Where the UIC, UFD, or directory is optional, two sets of brackets are shown (for example, [[g,m]]).
. . .	A vertical ellipsis shows where elements of command input or statements in an example or figure have been omitted because they are irrelevant to the point being discussed.
KEYNAME	This typeface denotes one of the keys on the terminal keyboard; for example, the RETURN key.
"print" and "type"	The term "print" refers to any output sent to a terminal by the system. The term "type" refers to any user input from a terminal.
black ink	In examples, what the system prints or displays is printed in black.
red ink	In interactive examples, what the user types is printed in red. System responses appear in black.

Summary of Technical Changes

The following sections list features, qualifiers, and restrictions that are new to the error logging system or have been modified for the RSX–11M–PLUS and Micro/RSX Version 4.0 operating systems. These new or modified features are documented in this revision of the *RSX–11M–PLUS and Micro/RSX Error Logging Manual.*

New Hardware Support

RSX–11M–PLUS and Micro/RSX Version 4.0 support the following new hardware:

- The PDP–11/73, PDP–11/83, and PDP–11/84 processors
- The RA70, RD52, RD53, RD54, and RX33 disk drives
- The TK50 and TU81E tape drives

New or Modified Features

The error logging system has the following new or modified features:

- Internal I/O Features
- Non-DIGITAL Device Recognition
- Error Logging Control Files
- DCL Support for ELI and RPT Commands
- Independent Hard and Soft Limits
- New Control File Modules

Internal I/O Features

The error logging system supports error logging for internal I/O operations such as data caching.

Non-DIGITAL Device Recognition

You no longer need to add a record to the DEVICE_INFO table in the DEVSM1 module for the error logging system to recognize a user-written device level module. After you write the device level module for your devices, compile the module or modules with the DSP2P1.SYM file. Next, insert your module in ERRLOG.ULB. The name of a user-written module must be in the following form:

```
ExxUSR
```

The letters xx stand for the device mnemonic. Your device mnemonic cannot be the same as any DIGITAL-supplied device mnemonic.

Specify your module name in the MODULE statement as follows:

```
MODULE ExxUSR
```

In your module, you must set the variable INDICATE.TAPE_FLAG to false if the device you specify is a disk.

If the device you specify is not a disk, set the variable INDICATE.TAPE_FLAG to true.

Remember to set your INTERMOD_DEVERR variables accordingly.

If there is a NOTES module, its name must be in the following form:

```
NxxUSR
```

DCL Support for ELI and RPT Commands

The ELI and RPT commands can be entered at the DCL level. This manual documents the DCL command equivalent for each MCR-level ELI and RPT command.

Independent Hard and Soft Limits

Error logging allows hard and soft error limits to be reached independently. Previously, reaching one of the limits would disable logging of either kind of error on that device. Now, reaching the soft limit does not affect the logging of hard errors and vice versa.

Device timeouts are logged as hard errors if unrecoverable and as soft errors if recoverable.

New Control File Modules

The error logging system contains the following new control file modules:

- The DSP8P1 dispatcher module for CPU-detected errors
- The E118x CPU-level module to process memory parity errors for the PDP–11/73, PDP–11/83, and PDP–11/84 processors

New or Modified Qualifiers

The DCL ELI command SHOW ERROR_LOG displays error logging information for the specified device or, if you do not specify a device, displays error logging information for all devices.

Micro/RSX supports the following qualifiers for the SHOW ERROR_LOG command:

- /CURRENT
- /HISTORY
- /NEW

The qualifiers for the SHOW ERROR_LOG command have the following functions:

/CURRENT

Displays errors that have occurred on all devices since the last time the error logging system was started using the /UPDATE or /ZERO switch. This qualifier provides the same functionality as the MCR ELI/SH/CU command.

/HISTORY

Displays a summary of errors that have occurred on all devices since the last time the error logging system was started using the /ZERO qualifier. This qualifier provides the same functionality as the MCR ELI/SH/HI command.

/RECENT

Displays a brief description of errors (in chronological order) that have been recorded by the error logging system. This qualifier provides the same functionality as the MCR ELI/SH/NE command.

New Error Messages

The Report Generator task (RPT) uses the following new error messages:

- ERLCNF-F-ILLPACSPC
- ERLRPT-F-FORINVCHR

ERLCNF-F-ILLPACSPC

ERLCNF-F-ILLPACSPC occurs when you use an invalid packet specification with an RPT report-generating command.

ERLRPT-F-FORINVCHR

ERLRPT-F-FORINVCHR occurs when a control file module executes a WRITE or WRITE_GROUP statement with a !DP directive that contains an invalid character.

Restriction

Error logging has the following restriction:

Data Subpacket Address Restriction

Because the error logging routines have been moved to the Executive common, the following restriction now applies: if a driver of an error logging device calls the $CRPKT routine to create an error logging packet, the data address for the data subpacket must not be an address within the driver. Specifically, the address must not be mapped by APR 5, as this APR is used to map the common. Any user-written driver that performs such a function must allocate a piece of pool, fill in the appropriate information, and pass the pool address to the Create Packet routine.

Chapter 1
Introduction

The RSX-11M-PLUS and Micro/RSX error logging system records information about errors and events that occur on your system hardware, either for immediate action or for later analysis and reporting. Error logging handles mass storage device (disk and tape) errors, as well as memory errors. Since error logging is a part of the RSX-11M-PLUS and Micro/RSX operating systems, it is most effective for hardware errors that allow the system to continue functioning.

Error logging is not used to detect information about operating system failures or about device problems that cause the system to fail. However, it does provide information about what I/O activities occurred on a device at the time of an I/O failure, and it detects errors in internal I/O operations such as data caching. If your system includes the Crash Dump Analyzer (CDA), CDA can provide reports on operating system failures.

You can use error log reports to determine that a device is having problems before the device actually fails and causes you to lose data. For example, a report showing a pattern of recurring errors from different blocks on a single disk head may indicate that the head needs to be replaced.

1.1 Error Logging Operation

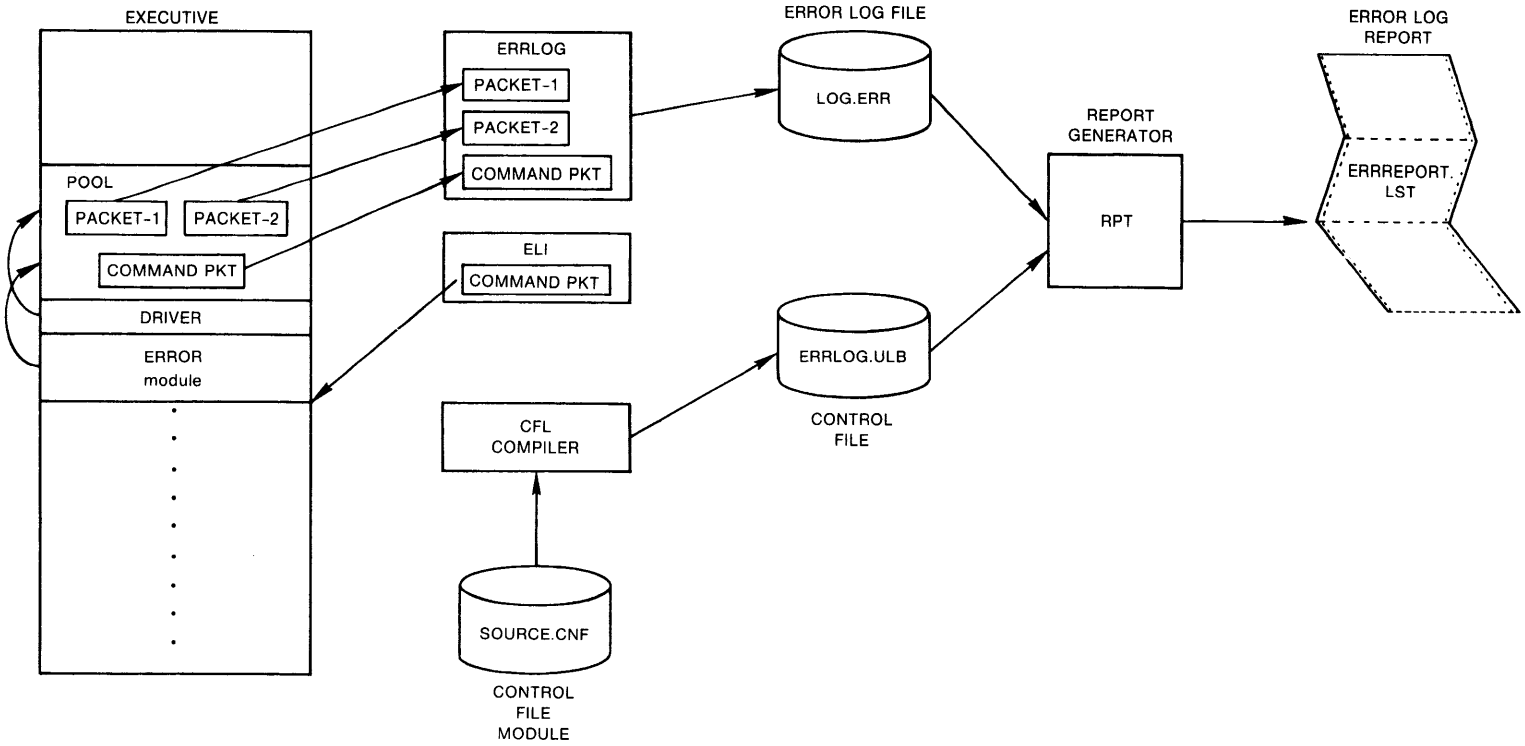
The complete error logging system is composed of the following four tasks:

- The Error Logger (ERRLOG)
- The Error Log Interface (ELI)
- The Report Generator (RPT)
- The Control File Language (CFL) compiler

When the Executive or a device driver detects an error, Executive routines create an error log packet in pool to describe the event. (See Appendix C for a description of the error log packet.) ERRLOG then writes the packet from pool into the error log file on disk, usually within a few seconds of when the packet is created.

Figure 1-1 shows the interaction of the error logging system tasks with routines in the Executive.

Figure 1-1: Error Logging System



ERRLOG receives user commands from the Error Log Interface (ELI) to control ERRLOG operation. These commands send error log packets called command packets to the ERRLOG task.

The Report Generator (RPT) generates reports from the information in the error log file.

RPT uses a library of modules written in the Control File Language (CFL) to interpret data from the error log file and from user commands. The CFL compiler is also part of the error logging system. You can use CFL to recompile DIGITAL-supplied control file modules to include patches to the modules supplied in the future. You can also use CFL to create and compile control file modules for devices other than those DIGITAL supplies. Chapter 4 explains the control file module architecture and includes annotated DIGITAL control file modules. Chapter 5 documents the Control File Language.

1.1.1 Executive Routines

Whenever the RSX–11M–PLUS or Micro/RSX system is running and error logging is active, routines in the Executive collect information from device drivers and other tasks and write the information into error log packets in system pool.

The Executive gathers information on the state of the registers when a device error occurs, and includes information on system events such as device mounts and dismounts. You can also insert a text message into the error log file using the MCR command System Service Message (SSM) or the DCL command MESSAGE/ERROR_LOG. (See the *RSX–11M–PLUS MCR Operations Manual.*)

If error logging is not active on the system, the device drivers still detect each hardware error, but the Executive does not create error log packets.

The error logging system makes a distinction between hard errors and soft errors. Hard errors are those that cause an I/O operation to be aborted because the device driver cannot recover from the error. The task that issued the I/O request receives an error code indicating that the operation failed. Soft errors are those from which the device driver can recover. The task that issued the I/O request does not receive an error notification because the request eventually succeeds.

The error logging system logs both hard and soft errors. Thus, you can have a system functioning properly, with no errors reported to any tasks in the system, with errors still being encountered and logged. For this reason, error logging terminology sometimes refers to errors as events; they do not always mean an actual failure.

When error logging is active, the Executive writes the data from a single event into one error log packet and assigns a sequence number unique to that event to the packet. The Resource Monitoring Display (RMD) shows the highest assigned sequence number as ERRSEQ, the total number of errors since error logging operations began.

When ERRLOG writes the packet in a file, the packet gets a number that describes its location in the file relative to other packets. RPT uses this number to refer to the event in later operations. The number does not change unless the organization of the file changes. For example, if an earlier error log file is appended to the current error log file, the packet numbers in the appended file will change.

Thus, you can generate a brief format RPT report to determine the packet numbers of the most significant errors on your system, and then generate a full format report, by packet number, of only those errors.

The Executive includes a directive for error logging (SMSG$) that sends error log packets directly to the Error Logger. (See the *RSX-11M-PLUS and Micro/RSX Executive Reference Manual* for an explanation of how to use this directive.) User tasks can use SMSG$ to communicate with the Error Logger.

1.1.2 ERRLOG and ELI

ERRLOG writes the error log packets from pool to the error log file in binary format. Only RPT can interpret and format data from the error log file.

To issue a command to ERRLOG, type an ELI command to perform one of the ERRLOG functions (logging, limiting, or file naming). ELI sends an error log command packet to ERRLOG with instructions on the function to be performed, and ERRLOG returns the results, if any, to ELI.

The ERRLOG task allows you to specify two files to contain the error log packets written to disk. ERRLOG uses the first file, the error log file, unless an error is detected while ERRLOG writes to the file. If an error is detected, ERRLOG switches to the second file, the backup file. ELI commands allow you to establish or change the names of the error log file and backup file.

The error logging system automatically limits the number of events it logs on a given device. This error limit can be changed dynamically by ELI commands while error logging is running. The system uses limits in case the device starts to accumulate a large number of errors. Without limits in these cases, the error log file would quickly become large and difficult to analyze. Error limiting does not lose useful information because, usually when a large number of events occurs on a device, most of them are the same and you can generalize from a report on a small number of the events.

After a device reaches a particular error limit, logging of that type of error on the device stops until you reset the error count to zero or raise the error limit.

ERRLOG sends a message to the console terminal or to any terminal that has allocated the device, explaining that the device reached the error limit. Error limiting does not affect the operation of the device itself; it only starts or stops error logging on the device.

1.1.3 RPT

RPT creates reports on the data in the error log file, based on information in the error log control file and commands supplied by the user. Modules in the error log control file tell RPT how to interpret and print entries from the error log file for a specific operating system.

When you are ready to generate an error log report, you can run RPT to select the information you want to include in the report. RPT can generate reports in brief and full format on any collection of error log packets you select. For example, you can select reports on a specific device by device name, device type, volume label, pack identification, or drive serial number. You can also select reports of a specific error type or you can select a full report of all the error log packets in the error log file.

Error log reports can contain both context information and device-supplied information.

Context information, which appears in full format reports, contains operating system version information and some information about the CPU model. Context information on the I/O operation that encountered the failure is recorded for device errors. This information allows you to correlate events recorded in the error log file with other events in the system. For example, hard I/O errors often cause the task issuing the I/O request to exit with an error, since many tasks cannot recover from I/O errors. You can also use information about the I/O operation to determine what operation the device driver attempted at the time of the failure.

Error logging records information on each I/O operation that takes place when an error occurs on other system error logging devices. This information is useful if you have a failure that may be related to interactions between devices. For example, I/O bus timing problems may show up as a problem only when more than one I/O operation takes place at the same time on the system.

In a full report, RPT also includes all device-supplied information, including the status of registers and any other information the device provides. Each device supplies one or more machine words of information when an error occurs. RPT decodes each item of device information according to the terminology used in the device maintenance manual. Additional information may be listed in parentheses to help you understand the significance of a decoded item.

In RPT reports, decoded items that are abnormal are flagged with an asterisk (*). These items may or may not represent error conditions, depending on the state of the device. You should interpret an asterisk flag as a "look at me first" message. RPT reports flag more than one item on most devices.

RPT reports also flag more than one item if a device encounters an error or cannot perform an operation because of another error condition. This condition occurs when an abnormal device status condition causes an I/O function to fail. The RPT report flags both the I/O function failure and the abnormal device status.

An error definition in the RPT report then compresses all the device-supplied information to a single item reflecting the most probable error reported by the device.

1.1.4 CFL

The error logging system includes a Control File Language (CFL) compiler that is used to recompile patched DIGITAL-written control file modules or user-written modifications or additions to modules. Chapters 4 and 5 describe the operation of the CFL compiler and the DIGITAL-supplied control file module for the RM02/RM03 disk drive.

1.2 Error Logging Options

Routines in the Executive respond to the following four types of errors:

- Unexpected traps or interrupts
- Device errors
- Device timeouts
- Memory errors

1.2.1 Unexpected Traps or Interrupts

When your system includes error logging support, all unused system vectors are filled with pointers to routines in the Executive. Therefore, routines in the Executive are called if a trap or interrupt occurs to one of these unused vectors. For example, a noisy electrical environment or a static discharge may cause an unexpected trap or interrupt to one of the unused vectors, or a valid interrupt may be vectored to the wrong address. In these cases, the Executive records this information.

1.2.2 Device Errors

Device errors are problems that a device encounters while carrying out a software-requested operation. When a device error occurs and error logging is active, the device driver calls Executive routines to record the contents of the device registers or other hardware-supplied information. The registers indicate the state of the device and its controller. The routines also record information found in the actual I/O request to the driver, such as the type of operation attempted. This information can help you interpret the device error.

1.2.3 Interrupt Timeouts

Interrupt timeouts occur when the device that initiated an operation fails to complete the operation within the length of time the driver specified. When the transfer starts, software timers detect interrupt timeouts. The system records the same information for timeouts that it records for device errors.

1.2.4 Memory Errors

Memory errors occur when the parity bit stored with the data during a write operation does not match the parity calculated when the data is read. Some types of main memory use parity to ensure integrity of the information. All RSX–11M–PLUS systems (except the pregenerated systems and Micro/RSX) include support for logging memory parity errors.

Chapter 2

Error Logger Task (ERRLOG) and Error Log Interface (ELI)

This chapter describes how to use the Error Logger task (ERRLOG) and the Error Log Interface (ELI). Chapter 1 provided a general overview of how ERRLOG and ELI work, along with the Report Generator (RPT), to form the complete error logging system.

ERRLOG gets event and status information from device drivers and the Executive in the form of error log packets and writes the packets in an error log file on disk. The Executive also performs error limiting to allow a maximum number of errors to be logged on each device before logging stops.

ELI, the user interface to ERRLOG, includes switches to perform the following operations:

- Start or stop logging or limiting
- Change device error limits or error counts
- Establish or change log file names
- Display information about the error logging status of any device or of the entire system

ERRLOG is the only part of the error logging system that must be installed for error logging to occur. You can install ELI when you issue commands to ERRLOG and install RPT when you create reports.

MCR and DCL Equivalents

If your system supports DCL, you can use DCL error logging commands in place of most MCR ELI switches. Where ELI commands or their formats are listed in this chapter, the MCR form is listed first and the DCL equivalent is listed directly underneath. For example, the following format line shows an MCR ELI command, and then the DCL command that performs the same operation.

Format

```
[filespec]/LOG [/subswitch[es]]
START/ERROR_LOG [/qualifier[s]] [filespec])
```

2.1 Installing ERRLOG and ELI

To install the ERRLOG task, enter the following command from a privileged terminal or as an entry in the system startup command file (both the MCR and DCL commands are shown):

```
>INS $ERL [RET]
$ INSTALL $ERRLOG [RET]
```

To install ELI, enter the following command from a privileged terminal or as an entry in the system startup command file:

```
>INS $ELI [RET]
$ INSTALL $ELI [RET]
```

To invoke ELI after it is installed, issue the following MCR command from any terminal:

```
>ELI [RET]
```

If ELI is not installed, you can invoke it from a privileged terminal using the following command:

```
>RUN $ELI [RET]
$ RUN $ELI [RET]
```

When ELI is invoked, your terminal will display the following prompt:

```
ELI>
```

2.2 Using ERRLOG and ELI

You can use the ELI /SH switch to display error logging information from any terminal. However, you must use a privileged terminal to execute any other ELI commands.

Enter each command on a separate line unless the command description specifies otherwise.

The general MCR format of an ELI command is as follows:

```
ELI>filespec [device:[:...]]/switch[/...]
```

The general DCL format of an ELI command is as follows:

```
$ command [/qualifier[s]] [device[s]] filespec
```

Parameters

filespec
The name of an error log file or file to append to the current error log file.

device
A device mnemonic indicating the device for which you want to perform error logging.

switches
Switches to set, change, or display ERRLOG operation. (You must specify at least one switch on each ELI command line.)

If you want to start error logging and use only the ERRLOG defaults, enter the following ELI command:

```
ELI>/LOG [RET]
```

```
$ START/ERROR_LOG [RET]
```

This command starts ERRLOG, using LB:[1,6]LOG.ERR as the default log file and LB:[1,6]BACKUP.ERR as the default backup file. Specify the /LOG switch to use ERRLOG defaults.

The /LOG switch also starts error limiting to limit the number of hard and soft errors ERRLOG records on each device before it stops logging on that device. The default error limit, used when you begin limiting with the /LOG switch, is five hard errors and eight soft errors for each device. You can change these limits with the /HL or /SL switches described in Section 2.4.1. However, you cannot use the switches to change limits on the same command line as the /LOG switch.

2.3 ELI Switches

This section describes the following four types of ELI switches and subswitches:

- Logging switches
- Error limiting switches
- File-naming switches
- The display switch

Remember that these switches only control operation of the Error Logger. Chapter 3 describes the RPT commands that generate actual error log reports. Chapter 5 describes the commands that control the Control File Language (CFL) compiler.

Table 2–1 summarizes the ELI switches and subswitches and their equivalent DCL commands. ELI syntax requires that you specify at least two characters of a switch name and as many additional characters as it takes to make the switch unique. Refer to the ELI switch descriptions in this section for the exact syntax and use of each command.

Table 2–1: ELI Switches and Subswitches

MCR ELI Switches and Arguments	DCL Commands and Qualifiers	Function
/LOG	START/ERROR_LOG	Begins error logging operation
/TY:arg[s]	/INCLUDE=arg[s]	Specifies type of error logging
ALL	ALL	Includes all information in error log file
C	CONTROL	Includes ELI control packets
E	ERRORS	Includes peripheral, processor, and memory errors
M	MEMORY	Includes errors in memory
PE	PERIPHERAL	Includes information from peripherals
PR	PROCESSOR	Includes events in CPU
SY	SYSTEM_INFORMATION	Includes system information, not device-specific
/NV	/NEW_VERSION	Creates a new file version
/-LIM	/NOLIMITING	Starts logging without error limiting
/UP[DATE]	/UPDATE (Micro/RSX only)	Updates ERRORS.ACC file
/ZE[RO]	/ZERO (Micro/RSX only)	Clears ERRORS.ACC file
/NOLOG	STOP/ERROR_LOG	Stops all error logging and limiting
/[-]LIM[IT]	SET ERROR_LOG/[NO]LIMITING	Starts or stops use of error limiting
/HL	SET ERROR_LOG/HARD_LIMIT	Sets limit for hard errors
/SL	SET ERROR_LOG/SOFT_LIMIT	Sets limits for soft errors
/RE[SET]	SET ERROR_LOG/RESET_COUNTS	Resets QIO and error counts to 0
/AP[PEND]	APPEND ERROR_LOG	Appends specified file to error log file
/DE[LETE]	/DELETE	Deletes specified file after appending
/SW[ITCH]	SET ERROR_LOG/NEW_LOG_FILE	Copies current file to new file
/DE	/DELETE	Deletes old logging file
/NV	/NEW_VERSION	Creates new version of logging file
/BA[CKUP]		Sets name of backup file to next-highest version of named file
/SH[OW]	SHOW ERROR_LOG	Displays error logging on all devices
filespec	/OUTPUT	Writes display to a file

Table 2–1 (Cont.): ELI Switches and Subswitches

MCR ELI Switches and Arguments	DCL Commands and Qualifiers	Function
/CU[RRENT]	/CURRENT	Shows error counts since last time /UPDATE was used (Micro/RSX only)
/HI[STORY]	/HISTORY	Shows error counts from ERRORS .ACC (Micro/RSX only)
/NEW	/RECENT	Shows most recent errors (Micro/RSX only)

2.3.1 The Logging Switches

ELI logging switches start or stop logging on all error logging devices in the system. (See Table 2–2, Error Logging Devices.)

2.3.1.1 The /LOG Switch

Format

```
[filespec]/LOG [/subswitch[/...]]
START/ERROR_LOG [/qualifier[/...]] [filespec]
```

The /LOG switch begins error logging operation and optionally allows you to specify a file in which the Error Logger writes the data it collects (see Section 2.4.2). If you specify an existing file, the /LOG switch appends new data to that file unless you also specify the New Version switch (/NV) in the command line.

The /LOG switch also turns on error limiting by default, unless you specify the No Limiting (/-LIM) switch to override it.

You can use the following subswitches with the /LOG switch:

/-LIM[IT]
/NOLIMITING

The No Limiting (/-LIM) subswitch begins logging with no error limiting. This subswitch overrides the default ERRLOG operation in which /LOG automatically turns on error limiting.

/NV
/NEW_VERSION

The New Version (/NV) subswitch causes the Error Logger to create a new version of the error log file (either the file you specify in the command line or the default error log file, LB:[1,6]LOG.ERR). This subswitch overrides the default operation in which the /LOG switch appends data to the current version of the error log file.

/TY[PE]:arg[,...]
/INCLUDE=[arg[,...]]

The /TYPE subswitch selects error log packets based on their packet type. The packets included in the report are determined by the following arguments:

MCR Arguments	DCL Arguments
ALL	ALL
C[ONTROL]	CONTROL
E[RROR]	ERRORS
M[EMORY]	MEMORY
PE[RIPHERAL]	PERIPHERAL
PR[OCESSOR]	PROCESSOR
SY[STEM_INFO]	SYSTEM_INFORMATION

The /TYPE subswitch arguments are described as follows:

ALL
Selects all error log command packets in the error log file. This is the default.

CONTROL
Selects error log command packets that control the Error Log Interface task (ELI).

ERROR
Selects error log packets from system peripherals, the processor, and memory.

MEMORY
Selects error log packets from events that occur in memory (such as memory parity errors).

PERIPHERAL
Selects error log packets from all peripheral devices that support error logging. This argument does not display system information (such as mounts and dismounts) for the devices.

PROCESSOR
Selects error log packets from events that occur in the CPU, such as unknown interrupts.

SYSTEM_INFO
Selects error log packets from events that occur on the system but are not specifically tied to errors on a single piece of hardware, such as time changes and system service messages, as well as some device-specific events such as mounts and dismounts.

/UP[DATE]
/UPDATE (Micro/RSX only)

The /UPDATE subswitch updates the historical part of the file of accumulated errors (ERRORS.ACC) to reflect the latest information in ERRORS.LOG. It then creates a new version of ERRORS.LOG that contains no error events. You cannot use the /UPDATE subswitch on the same command line with the /ZERO subswitch.

/ZE[RO]
/ZERO (Micro/RSX only)

The /ZERO subswitch erases all the accumulated errors in ERRORS.ACC. It then creates a new version of ERRORS.LOG that contains no error events. You cannot use the /ZERO subswitch on the same command line with the /UPDATE subswitch.

2.3.1.2 The /NOLOG Switch

Format

```
/NOLOG or /-LOG

STOP/ERROR_LOG
```

The /NOLOG switch stops error logging and error limiting on all devices.

2.4 Error Logging Devices

Table 2–2 lists the device modules included in the original version of the file LB:[1,6]ERRLOG.ULB, as distributed with the error logging system. However, if you have deleted any device modules from the universal library using the indirect command file described in Appendix A, your system will not include support for those devices. If you want error logging support for the devices listed in Table 2–2, include the control file module listed with the device in the universal library. See Appendix A for information on how to include and delete modules from the universal library.

Table 2–2: Error Logging Devices

Device	Control File Module
ML11	EML11
RA60	MSCP60 MSCPAT MSCPCE MSCPEN MSCPTO
RA80 /RA81	DEVUDA MSCP5X MSCPAT MSCPCE MSCPEN
RC25	MSCPSD MSCPAT MSCPCE MSCPEN MSCPTO DEVUDA
RD50 /RD51 /RD52	MSCPTO
RD53 /RD54	DEVUDA

Table 2-2 (Cont.): Error Logging Devices

Device	Control File Module
RK03/RK05	ERK05
RK06 /RK07	ERK67
RL01 /RL02	ERL12
RM02 /RM03	ERM23
RM05	ERM05
RM80	ERM80
RP02 /RP03	ERP23
RP04 /RP05 /RP06	ERP456
RP07	ERP07
RS11	ERS11
RS03 /RS04	ERS34
RX01	ERX01
RX33 /RX50	MSCP5X
TA11	ETA11
TC11	ETC11
TK50	ETK50
TS03 /TE10 /TT10	ET0310
TS11 /TU80	ETS11
TSV05	ETSV05
TU16 /TE16 /TU45	ET1645
TU58	ETU58
TU60	ETU60
TU77	ETU77
TU81E	ETU81

2.4.1 The Error Limiting Switches

The limiting switches control the error limiting operation of ERRLOG. You can use them to start or stop error limiting or to change error limits on specific devices. When a device reaches the user-specified error limit or the default error limit, ERRLOG displays the following warning message on the console terminal or on any terminal that has allocated or attached the device:

```
ERRLOG - - **WARNING: Device dd: Exceeded (xxxx) Limit (n)
```

In the warning message, xxxx is the type of limit (hard or soft) and n is the number to which the limit is set.

When the device reaches an error limit, error logging for that type of error stops on the device until you reset the error and QIO counts to 0 or raise the error limit.

You can reset the error and QIO counts to 0 with the ELI /RESET switch. Mounting or dismounting the device or rebooting the system also resets the error and QIO counts to 0. However, using the /-LOG switch to stop logging does not reset the error and QIO counts.

Logging on a device stops only when the device reaches both of the limits set for hard and soft errors. If, for example, the device reaches its limits for hard errors but not for soft errors, it will continue to log soft errors until the soft error limit is also reached.

2.4.1.1 The /LIMIT Switch

Format

```
/[-]LIM[IT] or /[NO]LIM
```

```
SET ERROR_LOG/[NO]LIMITING
```

The /LIMIT and No Limit (/-LIM) switches start and stop the use of error limits, respectively. These limits are set by default for all devices on the system when you enable error logging or they are set for individual devices with the hard and soft limit switches described below. The /LIM switch does not activate error logging if it is not currently active on the system.

When you specify the /LOG switch to begin error logging, it automatically starts error limiting on all error logging devices unless you inhibit limiting with the /-LIM switch.

The /-LIM switch inhibits the incrementing of hard and soft error counts, which are displayed by the ELI /SH switch and in the RMD "I" page.

Note

If you start error logging with no limiting and the error counts happen to be at zero when error logging is started, ELI displays the counts as zeros even though errors continue to be logged. To check the number of errors that occur when limiting is off, you must use RPT to generate an error report.

2.4.1.2 The Hard Limit Switch

Format

```
ddnn:[,...]/HL:n[/SL:n]
```

```
SET ERROR_LOG/HARD_LIMIT:n[/SOFT_LIMIT:n] ddnn:[,...]
```

The Hard Limit (/HL) switch sets limits for the number of hard errors that error logging records on the device specified. Hard errors occur on a device when an I/O operation fails and cannot be recovered by the device driver. You can set hard error limits for more than one device in the same command line, as long as the limits are the same. The default hard error limit on each device is five.

The value for n can be 0 to 255. If you set the limit to 255, logging continues without stopping (the limit is infinite). If the limit is set to 0, no errors will be logged.

You can use the Soft Limit (/SL) switch on the same command line with the /HL switch, thereby controlling both hard and soft device error limits on the same command line. You can specify different limits for hard and soft errors.

2.4.1.3 The Soft Limit Switch

Format

```
ddnn:[,...]/SL:n[/HL:n]
```

```
SET ERROR_LOG/SOFT_LIMIT:n[/HARD_LIMIT:n] ddnn:[,...]
```

The Soft Limit (/SL) switch sets limits for soft errors. Soft errors occur on a device when an I/O operation fails, but succeeds in a subsequent retry attempt. You can set soft error limits for more than one device in the same command line, as long as the limit is the same. The default soft error limit for each device is eight.

The value for n can be 0 to 255. If you set the limit to 255, logging continues without stopping (the limit is infinite). If the limit is set to 0, no errors will be logged.

You can use the Hard Limit (/HL) subswitch on the same command line with the /SL switch, thereby controlling both hard and soft device error limits on the same command line. You can specify different limits for hard and soft errors.

2.4.1.4 The /RESET Switch

Format

```
ddnn:[,...]/RE[SET]
```

```
SET ERROR_LOG/RESET_COUNTS ddnn:[,...]
```

The /RESET switch resets the QIO count and error count for the specified devices to 0. You can specify up to 14 devices in one command line. You cannot reset QIO and error counts on all devices in the system at once by specifying the /RE switch without specifying devices.

When ERRLOG resets the counts to 0, it displays the following message on the console terminal:

```
ERRLOG - - Error and QIO counts reset for ddnn:
```

2.4.2 The File-Naming Switches

The file-naming switches establish and change the names of error log files.

2.4.2.1 The /LOG Switch

Format

```
[filespec]/LOG
```

```
START/ERROR_LOG [filespec]
```

The /LOG switch, which also initializes the Error Logger, sets the name of the error log file that the Error Logger uses. If you specify an existing error log file, the default operation is to append data to the current version of that file. To override the default, specify the New Version (/NV) subswitch. The Error Logger then creates and writes data in a new version of the file. The /LOG switch does not work when error logging is already active on your system.

The default error log file specification is LB:[1,6]LOG.ERR. The /LOG switch also specifies LB:[1,6]BACKUP.ERR as the backup file.

2.4.2.2 The /APPEND Switch

Format

```
filespec/AP[PEND][/D[ELETE]]

APPEND/ERROR_LOG[/DELETE] filespec
```

The /APPEND switch appends the specified file to the end of the current log file. Error logging must be active for this switch to work.

The default operation is to append the specified file to the current error log file and to keep the appended file.

You can use the following subswitch with the /APPEND switch:

/DE[LETE]
/DELETE
The /DELETE subswitch causes the Error Logger to delete the specified file after it is copied to the end of the current error log file.

2.4.2.3 The /SWITCH Switch

Format

```
filespec /SW[ITCH][/subswitch]

SET ERROR_LOG /NEW_LOG_FILE:filespec [/qualifier]
```

The /SWITCH switch copies the current error log file to the file you specify and begins logging in that file. The default operation appends data to an existing version of the file and preserves the old version of the error log file.

You can use the following ELI subswitches or DCL qualifiers on the command line with the /SWITCH switch:

/NV
/NEW_VERSION
The New Version (/NV) subswitch creates a new version of the file you specify. This subswitch overrides the default operation in which the /SW switch appends data to the latest version of the file.

/DE[LETE]
/DELETE
The /DELETE subswitch causes the Error Logger to delete the current error log file after it copies the file to the new file you specify.

2.4.2.4 The /BACKUP Switch

Format

```
filespec /BA[CKUP]
```

The /BACKUP switch specifies the file to be used as a backup file if the Error Logger cannot write to the current log file. By default, the backup file is LB:[1,6]BACKUP.ERR.

The backup file specification is kept, but no file is created until needed. You may wish to have your backup file on a different device from the current log file. By default, both files are on pseudo device LB:.

When the Error Logger cannot write to the current log file, it creates and opens the backup file and writes to it. At that point, you no longer have a backup file, and the Error Logger displays the following message on the console terminal:

```
ERRLOG - - Log file error - logging continuing on backup file
```

After error logging switches to the backup file, there is no longer a backup file available.

The Error Logger uses the specified backup file as the current error log file. It does not rename the file to LOG.ERR, even though the file is now the error log file.

At this point, you should specify a new backup file, using the /BACKUP switch. Otherwise, if the Error Logger cannot write to the new log file, it will not be able to continue by writing in a backup file.

If the Error Logger tries to switch logging to a nonexistent backup file, it displays the following message:

```
ERRLOG - - Backup file error - logging discontinued
```

When that happens, logging stops and must be restarted.

If you create the backup file on a disk other than the disk containing the error log file, this ensures that logging will continue even if the disk with the error log file develops problems.

2.4.3 The /SHOW Switch

Format

```
[filespec=] [ddnn:[,...]]/SH[OW][/subswitch]

SHOW ERROR_LOG[/qualifier[/...]] [ddnn:[,...]]
```

The Display (/SHOW) switch provides a brief display of error logging information on the devices specified (up to 14). The default is to display this report on your terminal. If the command does not specify devices, the Error Logger displays information on all error logging devices in the system by default.

The display shows the total number of errors that have occurred since error logging was started in the current ERRORS.LOG file. The display includes only errors from devices that are on line. It does not show the time or order in which the errors occurred.

The /SHOW switch uses the following arguments and subswitches:

filespec
/OUTPUT[:filespec]

The filespec argument allows you to specify that the output from the /SHOW switch be written in a file instead of displayed on your terminal. In DCL, if you use the /OUTPUT qualifier without a file specification, the system uses the file name ERRORS.LST. If you specify a file name, the system uses that name.

/C[URRENT](Micro/RSX only)
/CURRENT

The /CURRENT subswitch displays the error counts that have occurred on all devices since error logging was started using the /LOG/UPDATE switch. The /CURRENT subswitch displays the current section of the ERRORS.ACC file.

/H[ISTORY](Micro/RSX only)
/HISTORY

The /HISTORY subswitch displays the error counts from the historical part of the file ERRORS.ACC. This includes hard and soft errors from the last time the file was set to zero through the last time the file was updated. Errors that occurred after the file was updated are not included in the historical display, but rather in the current display. To see them, use the /CURRENT subswitch.

/N[EW](Micro/RSX only)
/RECENT

The /NEW subswitch displays the information in the ERRORS.LOG file to show only the most recent errors. It also shows the time and order in which they occurred. This display automatically includes all error logging devices.

Example 2–1 illustrates the output from the operation of the /SH switch.

Example 2–1: Error Logging Status

```
Error Logging Status    12-AUG-87 00:51:54

Logging: On     Limiting: On

Log File: LB:[1,6]LOG.ERR    File ID: DR3: 32,252

Backup File: LB:[1,6]BACKUP.ERR

     Device          Hard Error          Soft Error          QIO
     Name            Count/Limit         Count/Limit         Count

     MM0:            0./5.               0./8.               23.
     MM1:            0./5.               0./8.               9776.
     MM2:            0./5.               0./8.               0.
     MM3:            0./5.               0./8.               0.

     DB0:            0./5.               0./8.               14144.
     DB1:            0./5.               0./8.               0.
     DB2:            0./5.               * 8./8.             46528.

     DR0:            0./5.               0./8.               0.
     DR1:            0./5.               0./8.               0.
     DR2:            0./5.               0./8.               164234.
     DR3:            0./5.               0./8.               625364.

     DS:             0./5.               0./8.               130.
     DS1:            0./0.               0./0.               0. (Offline)

     DK0:            0./5.               0./8.               1.
     DK1:            0./5.               0./8.               0.

     DM0:            0./5.               0./8.               0.
     DM1:            0./5.               0./8.               0.

     DL0:            0./5.               0./8.               0.
     DL1:            0./5.               0./8.               0.

     DT0:            0./5.               0./8.               0.
     DT1:            0./5.               0./8.               0.
     DT2:            0./5.               0./8.               0.
     DT3:            0./5.

     DY0:            0./5.               0./8.               1.
     DY1:            0./5.               0./8.               1.

     DD0:            0./5.               0./8.               0.
     DD1:            0./5.               0./8.               0.
```

If you specify device names with the /SHOW switch, the output is the same as shown in Example 2–1, except that the display only includes information on the devices you specified.

The asterisk next to the soft error limit for DB2 indicates that DB2 reached the soft error limit and logging of soft errors stopped. Note that the logging of hard errors will continue on DB2 until the hard error limit is reached.

The display continues to record additional QIOs on the device, even after logging stops, because the Executive maintains the QIO count. Therefore, the ratio of errors to QIOs on the device does not necessarily give you an accurate statistical error percentage.

2.5 ERRLOG and ELI Messages

ERRLOG displays messages on the console terminal when errors occur during an operation. In some cases, ERRLOG displays messages on any terminal that has allocated or attached the device on which the error occurs. ELI displays messages on the terminal that invoked it. This section describes the messages, their causes, and possible user responses.

2.5.1 ELI Messages

ELI—ERRLOG not installed

Explanation: ERRLOG is not installed on the system.

User Action: Install ERRLOG from a privileged terminal and issue the ELI command again.

ELI—Failed to communicate with ERRLOG

Explanation: ELI could not communicate with ERRLOG using the Executive directive SMSG$.

User Action: Fatal error. No user action is possible.

ELI—File name must be specified

Explanation: You used a /BACKUP, /APPEND, or /SWITCH switch without specifying a file name.

User Action: Reenter the ELI command with an appropriate file specification.

ELI—Get Command Line error

Explanation: The Get Command Line procedure failed.

User Action: This may be a temporary condition. Retry the operation.

ELI—Illegal switch combination

Explanation: You used an ELI switch in combination with subswitches other than those allowed on a command string with that switch. (See Table 2–1.)

User Action: Reenter the command string, specifying a valid combination of switches on each string. Use a separate command string for additional switches, if necessary.

ELI—Maximum number of devices exceeded

Explanation: You attempted to reset QIO and error counts on more than 14 devices in one command string.

User Action: Specify the /RESET switch again, with 14 or fewer devices.

ELI—Switch requires device name ddnn: only

Explanation: You specified both a device name and a UFD and/or a file name with an ELI switch that accepts only a device name.

User Action: Reenter the command line, specifying only a device name.

ELI—Syntax error

Explanation: You used an invalid switch or file specification or made some other syntactical error.

User Action: Reenter the command line, using the proper command string syntax.

2.5.2 ERRLOG Messages

ERRLOG—Backup file error - logging discontinued

Explanation: ERRLOG encountered an error when it wrote in the log file. It then tried to write in the backup file, but could not. This error occurs if you fail to establish a new backup file after ERRLOG switches logging to the backup file.

User Action: Issue an ELI /BACKUP command to establish a new backup file and restart logging.

ERRLOG—Device not in system

Explanation: ERRLOG tried to use a device that is not in the system configuration.

User Action: Check to be sure you specified the correct device and reenter the command. If the device is correct, no user action is possible.

ERRLOG—Error and QIO counts reset for ddnn:

Explanation: The error and QIO counts for a given device were reset.

User Action: No user action is necessary. This is an informational message.

ERRLOG—Error log packet too long

Explanation: ERRLOG encountered an error log packet that was too large. The error log packet was corrupt.

User Action: If the error logging system includes user-generated error log packets, check the code to make sure none of the packets are too long. Otherwise, submit a Software Performance Report (SPR).

ERRLOG—Failed to assign LUN

Explanation: ERRLOG tried to assign a logical unit number (LUN) to a terminal to send a notification message and the assignment failed. This occurs when a device exceeds the error limit set for it and ERRLOG tries to notify the terminal or task that has the device allocated or attached.

User Action: No user action is necessary. The error limiting operation succeeded. This informational message tells you ERRLOG was unable to notify the allocating terminal.

ERRLOG—File I/O error

Explanation: ERRLOG tried to execute an ELI /SWITCH or /APPEND command and could not open the new file or copy the old file to the new one. When this error occurs, logging continues in the original log file.

User Action: No action is required to continue logging. Retry the ELI /SWITCH or /APPEND command.

ERRLOG—Log file error - logging continued on backup file

Explanation: An error occurred when ERRLOG tried to write in the error log file. The logging operation transferred to writing in the backup file. The backup file becomes the log file, but retains the given name.

User Action: Issue an ELI command to establish a new backup file. Otherwise, if ERRLOG gets an error when it writes in the new file (the previous backup file), it will not find a backup file to use.

ERRLOG—Logging already active

Explanation: ERRLOG received an ELI command to begin logging when logging was already running.

User Action: No user action is necessary to continue logging.

ERRLOG—Logging initialized

Explanation: ERRLOG displays this message on the console terminal when ELI starts ERRLOG operation using the /LOG switch.

User Action: No user action is necessary. This is an informational message.

ERRLOG—Logging not active

Explanation: The ERRLOG task is not currently running on your system.

User Action: Issue an ELI /LOG command from a privileged terminal and retry the operation.

ERRLOG—Logging stopped

Explanation: When ELI stops ERRLOG operation, using the /-LOG switch, ERRLOG displays this message on the console terminal.

User Action: No user action is necessary. This is an informational message.

ERRLOG—No data subpacket

Explanation: ERRLOG tried to use a corrupted data subpacket.

User Action: If the error logging system includes a user-written control file module to generate error log packets, check the code. Otherwise, submit an SPR.

ERRLOG—No device subpacket

Explanation: ERRLOG tried to use a corrupted device subpacket.

User Action: If the error logging system includes a user-written control file module to generate error log packets, check the code. Otherwise, submit an SPR.

ERRLOG—Privilege violation

Explanation: You tried to issue a privileged ELI command (to set or change ERRLOG operations) from a nonprivileged terminal. Nonprivileged users can only issue ELI /SHOW commands.

User Action: Log in to a privileged terminal and issue the commands.

ERRLOG—Task subpacket corrupted

Explanation: ERRLOG tried to use a corrupted task subpacket.

User Action: Submit an SPR.

ERRLOG—Unable to open file

Explanation: ERRLOG could not open the log file to begin logging. ERRLOG then transfers logging to the backup file immediately.

User Action: Issue an ELI command to establish a new backup file.

ERRLOG—Unknown command packet subtype

Explanation: ERRLOG encountered an unknown command packet subtype.

User Action: If the error logging system includes a user-written control file module to generate error log packets, check the code. Otherwise, submit an SPR.

ERRLOG—••WARNING: Device ddnn: exceeded xx Limit (x)

Explanation: Device ddnn exceeded the error limit set with an ELI Hard Limit (/HL) or Soft Limit (/SL) switch or the default error limit of five hard errors and eight soft errors.

User Action: Check to see if the number of errors indicates a serious hardware malfunction. To continue logging on the device, reset the QIO and error counts to zero with the /RESET switch or change the limits using the /HL or /SL switch.

Chapter 3
Report Generator Task (RPT)

This chapter describes how to use the Report Generator task (RPT) to create error log reports.

Chapter 1 provided an overview of the interaction of elements in the error logging system (the error log control file and the Control File Language (CFL) compiler). The RPT switches described in this chapter use modules from the error log control file to determine how to interpret and format information from the error log file. (See Chapter 2 for a description of how the Error Logger creates the error log file.) RPT and modules in the error log control file work together to interpret the information in the error log file and define an event that occurs on a device. They do not analyze the event itself or attempt to diagnose hardware failures.

All RPT reports use the same entry number to refer to the same error log packet, so you can use RPT brief reports to isolate a device or specific events occurring on that device, and then specify entry numbers to generate a full report on only the specific events you want to look at in more detail. Note, however, that some ELI commands may change the packet number associated with an event. For example, appending a file to the error log file will change the packet numbers in the appended file.

MCR and DCL Equivalents

If your system supports DCL, you can use DCL ANALYZE/ERROR_LOG commands instead of MCR RPT switches. Where RPT commands or their formats are listed in this chapter, the MCR form is listed first and the DCL equivalent is listed directly underneath. For example, the following format line shows an MCR RPT command, and then the DCL command that performs the same operation:

Format

```
/DA[TE]:argument
ANALYZE/ERROR_LOG/qualifier
```

3.1 Installing and Running RPT

Since RPT is a nonprivileged task, anyone can use it to create error log reports when it is installed on the system. To install RPT, enter the following MCR or DCL command from a privileged terminal or as an entry in the system startup command file:

```
>INS $RPT [RET]
$ INSTALL $RPT [RET]
```

To invoke RPT after it is installed, issue the following MCR command from any terminal:

```
>RPT [RET]
```

If RPT is not installed, you can invoke it from any terminal, using the following MCR or DCL command:

```
>RUN $RPT [RET]
$ RUN $RPT [RET]
```

When RPT is invoked, your terminal will display the following prompt:

```
RPT>
```

3.2 Using RPT to Create Error Log Reports

The error log control file needs at least three types of information from RPT switches to generate error log reports, as follows:

- How to select which error log packets to analyze
- How to format the error log packets
- How to summarize the information from the error log packets

Switches on the RPT command line provide this information, which is independent of the file specification they accompany.

3.2.1 The RPT Command Line

This section describes the general MCR RPT command line format and the general DCL ANALYZE/ERROR_LOG command line format. Note that the default parameters of the MCR RPT command line and the DCL ANALYZE/ERROR_LOG command line are different. This does not prevent DCL users from duplicating any MCR command, but you should be aware that the following MCR and DCL command formats are not exactly equivalent.

The MCR RPT Command Line

The only element you must specify in an MCR RPT command line is the equal sign (=). All other file and switch specifications in the command line are optional.

The general MCR format of an RPT command line is as follows:

```
RPT>[reportfile][/switch[/...]]=[inputfile][/switch[/...]]
```

reportfile

The name of the listing file that contains the error log report. If you do not specify the reportfile parameter, RPT sends the file by default to ERRREPORT.LST in your current directory.

Instead of a report file, you can specify TI: to send the report to your terminal. On systems with transparent spooling, you can specify LP: to send the report to the line printer.

switch(es)

Optional switches to control how RPT selects, formats, and summarizes information from the error log file. You can use the same switches with either the report file specification or the input file specification on the command line. For example, if your command line specifies "input file/WIDE," RPT will apply the wide format to the output file. In this case, the format width cannot apply to the input file. RPT uses the switches in the order you specify, but ignores which file specification they accompany.

input file

The only input file you can specify in the command line is the error log file, the disk file that the Error Logger creates in LB:[1,6]LOG.ERR.

RPT also uses a universal library of compiled control file modules as input. RPT looks first for the file LX:[1,6]ERRLOG.ULB. If it does not find it, RPT looks for the file LB:[1,6]ERRLOG.ULB. (Use pseudo device LX if you wish to save space on LB.) RPT includes this file by default and you cannot specify or change it from the command line, so it is not part of the format described previously.

RPT can, however, prompt for the name of a universal library. If you want RPT to prompt you for the universal library name, you must edit the RPTBLD.BLD file and make the value of USERCM nonzero, then relink RPT. If you do make this alteration, note that it has the additional effect of preventing you from issuing an RPT command line from the MCR level. Instead, invoke RPT as follows:

```
>RPT
CTL>universal library filespec
RPT>command line
```

The MCR RPT input and output files assume the defaults listed in Section 3.2.2, unless you specify otherwise in the command line.

The DCL ANALYZE/ERROR_LOG Command Line

The general DCL format of an RPT command line is as follows:

```
$ ANALYZE/ERROR_LOG [/OUTPUT[:reportfile]] [/qualifier[s]] [inputfile]
```

The default parameters are similar to those of the general MCR RPT command line, except for the reportfile parameter.

In DCL, you use the /OUTPUT qualifier to specify the report file. If you do not specify the /OUTPUT qualifier, the RPT report is displayed on your terminal in narrow format and no output file is created. If you use the /OUTPUT qualifier with no report file name specified, RPT creates a report in a file called ERRREPORT.LST. By default, the /OUTPUT qualifier creates a file in wide format.

3.2.2 Using the Default MCR RPT Command Line

To use the default RPT command line, enter the following MCR-level command:

```
RPT> = RET
```

The equal sign (=) causes RPT to use the file specification defaults and switch defaults listed in this section. This command creates a brief format report without any summaries, using all of the error log packets in the error log file. The file specification defaults are listed in Table 3–1.

Table 3–1: RPT File Specification Defaults

Parameter	Report File	Input File	Universal Library File[1]
Device	SY0	LB	LX,LB
UIC	Current UIC	[1,6]	[1,6]
File name	ERRREPORT	LOG	ERRLOG
File type	LST	ERR	ULB
Version	New	Latest	Latest

[1]Not specified by the user

The default RPT command line invokes the following switches:

/FORMAT:BRIEF
Creates a brief format report that contains one line for each error log packet described in the report. (See Section 3.3.2.)

/TYPE:ALL
Creates a report on packets describing all types of events: peripheral, processor, memory, control, and system information packets. (See Section 3.3.1.5.)

/DATE:RANGE:*:*
Creates a report on packets of all dates. (See Section 3.3.1.1.)

/DEVICE:ALL
Creates a report on all error logging devices in the system. (See Section 3.3.1.2.)

/PACKET:*:*
Creates a report for all packet numbers. (See Section 3.3.1.3.)

/SUMMARY:NONE
Does not create summary reports of statistical information on packets included in the report.

/WIDTH:WIDE
Creates a wide (132-column) report.

3.2.3 Using Multiple Arguments in RPT Command Lines

You can specify each RPT switch only once in a command line. However, some switches provide an alternative syntax that allows you to specify more than one argument for the switch, as follows:

```
/switch:(arg,arg[,...])
```

```
ANALYZE/ERROR_LOG/qualifier:(arg,arg[,...])
```

The parentheses, which are a required part of the command syntax, allow RPT to use more than one argument for the switch. If you do not specify the parentheses, RPT displays the following message on your terminal and exits:

```
ERLCNF-F-SYNTAXERR command line syntax error
```

As an example, to specify a report on more than one device, use the following RPT switch:

```
/DE:(DB,DM2:,DR)
```

```
ANALYZE/ERROR_LOG/DEVICES:(DB,DM2:,DR)
```

RPT generates a report on all the DB and DR devices in your system, as well as device DM2.

The switches that permit you to specify multiple arguments in this way are the following:

- The /DEVICE switch
- The /PACKET switch
- The /SERIAL switch (one drive and one pack serial number)
- The /TYPE switch
- The /SUMMARY switch

3.3 RPT Report Switches

This section describes the RPT switches according to the RPT requirement that they fulfill. These switches tell RPT how to perform the following tasks:

- Selecting packets
- Formatting packets
- Summarizing information from packets

RPT syntax requires only that you specify enough characters of a command or qualifier to make it unique. For example, you can specify /T for the MCR /TYPE switch, but you must specify /SU for the /SUMMARY switch to distinguish it from the /SERIAL switch.

The command line examples used throughout this chapter highlight the command they describe. Any switches not explained in the command descriptions assume the default values described in Section 3.2.2.

Table 3–2 summarizes the RPT report switches in alphabetical order.

Table 3–2: RPT Report Switches and Arguments

MCR RPT Switches and Arguments	DCL ANALYZE/ERROR_LOG Qualifiers and Arguments	Function
reportfile parameter	/OUTPUT	Specifies the output file for the RPT report
/DA[TE]:		Selects packet based on date
P:n	/PREVIOUS_DAYS:n	Specifies packets from previous n days
R	/SINCE:date/THROUGH:date	Specifies starting and ending dates
T	/TODAY	Specifies packets created on current day
Y	/YESTERDAY	Specifies packets created yesterday
/DE[VICE]:	/DEVICES:	Selects packets based on device
device name(s)	device name(s)	Specifies devices
ALL	ALL	Selects all devices on the system
/F[ORMAT]:		Describes how RPT formats the error log packets
B	/BRIEF	Displays each packet on one line
F	/FULL	Displays all of the information in the specified packet
N	/NODETAIL	Does not display information packet-by-packet
R	/REGISTERS	Displays the same information as the /FULL qualifier, but shows only the device registers on packets for peripheral errors
/P[ACKET]:	/ENTRY:	Selects packets based on packet number
start[:end]	start[:end]	Selects one or a range of packets
:	ALL	Selects all packets
/R[EPORT]:	/COMMAND:	Invokes a predefined string of switches
D	DAY	Displays full format reports based on current day's packets

Table 3–2 (Cont.): RPT Report Switches and Arguments

MCR RPT Switches and Arguments	DCL ANALYZE/ERROR_LOG Qualifiers and Arguments	Function
W	WEEK	Displays summary reports sorted by device error history based on previous seven days' packets
M	MONTH	Displays summary reports sorted by device error history based on previous 31 days' packets
SY	SYSTEM	Displays summary reports sorted by device error history for all packets in the system
user string	user string	Invokes a user-defined switch string
/SE[RIAL]:	/SERIAL_NUMBER:	Select packet based on serial number
D:n	DRIVE:n	Selects packet based on drive serial number
P:n	PACK:n	Selects packet based on pack serial number
D:n:P:n	DRIVE:n,PACK:n	Selects packets based on both drive and pack serial numbers
/SU[MMARY]:	/STATISTICS:	Selects the type of summary report
A	ALL	Selects all summary reports
E	ERROR	Creates a report based only on device errors
G	DISK_GEOMETRY	Creates a report based on disk geometry
H	HISTORY	Creates a report based on device error history
N	NONE	Creates no summary report
/T[YPE]:	/INCLUDE:	Selects packets based on packet type
A	ALL	Selects all packets in the error log file
C	CONTROL	Selects command packets from ELI
E	ERRORS	Selects packets from processor, memory, and peripherals
M	MEMORY	Selects packets from events in memory

Table 3–2 (Cont.): RPT Report Switches and Arguments

MCR RPT Switches and Arguments	DCL ANALYZE/ERROR_LOG Qualifiers and Arguments	Function
PE	PERIPHERAL	Selects packets from all peripheral devices
PR	PROCESSOR	Selects packets from events in CPU
S	SYSTEM_INFORMATION	Selects from system events, not hardware-specific
/V[OLUME]	/VOLUME_LABEL	Selects packets for peripheral errors based on the volume label
/W[IDTH]:	[NO]WIDE	Selects the width of RPT report
N		80 columns (ignored on summary reports)
W		132 columns

3.3.1 Packet Selection Switches

The following switches tell RPT how to select which error log packets to report on. (This selection is based on an attribute of the device or the packet or on the date and time that the packet was created.)

3.3.1.1 The /DATE Switch

Format

```
/DA[TE]:argument
```

```
ANALYZE/ERROR_LOG/qualifier
```

The /DATE switch allows you to select packets based on the date that an event occurred. This switch allows arguments to specify a range of dates or to specify a particular day. You can also use the /DATE switch with another switch, the /REPORT switch, to create reports for the previous week or month (see Section 3.3.4).

You can use the following arguments with the /DATE switch:

MCR Arguments	DCL Qualifiers
P[REVIOUS]:n days	/PREVIOUS_DAYS:n
R[ANGE]:start_date:end_date	/SINCE:start_date/THROUGH:end_date
T[ODAY]	/TODAY
Y[ESTERDAY]	/YESTERDAY

The /DATE switch arguments are described as follows:

PREVIOUS
Allows you to specify packets from the previous n days.

RANGE
Accepts starting and ending dates in the following standard RSX formats:

```
DD-MMM-YY

(DD-MMM-YY HH:MM:SS)
```

If you specify the second format, with time as well as date, the parentheses are a required part of the syntax.

When you use the starting date and ending date format, the starting date rounds off to a time of 00:00:00 and the ending date rounds off to 23:59:59.

The asterisk (*) used at the beginning of a range specification indicates any date through the specified ending date. For example, *:12-AUG-87 specifies all of the packets from the beginning of the error log file through August 12, 1987.

The asterisk used at the end of a range specification indicates any date since the specified beginning date. For example, 4-FEB-87:* specifies all of the packets from 00:00:00 on February 4, 1987, through the end of the error log file.

TODAY
Creates reports on error log packets created during the current day; that is, since midnight.

YESTERDAY
Generates reports on packets created during the previous day.

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /DATE switch as follows:

```
/DATE:RANGE:*:*
```

3.3.1.2 The /DEVICE Switch

Format

```
/DE[VICE]:device[s]=inputfile

ANALYZE/ERROR_LOG/DEVICES:device[s] [inputfile]
```

The /DEVICE switch allows you to select packets for a particular device, for more than one device, or for all the devices on the system. You can specify more than one device with the /DEVICE switch by using the special syntax described in Section 3.2.3.

MCR Arguments	DCL Arguments
device_list	device_list
ALL	ALL

The default argument is ALL.

RPT uses the following conventions for device names with the /DEVICE switch:

Mnemonic	Meaning
dd	Selects all devices with the mnemonic dd
ddnn:	Selects the device with the mnemonic dd and the unit number nn

For example, /DE:DM selects all DM devices, and /DE:(DM,DB2:) selects all DM devices and device DB2.

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /DEVICE switch as follows:

```
/DEVICE:ALL
```

3.3.1.3 The /PACKET Switch

Format

```
/P[ACKET]:packet_number[s]
```

```
ANALYZE/ERROR_LOG/ENTRY:packet_number[s]
```

The /PACKET switch allows you to select a packet or range of packets by specifying the packet identification numbers. You can determine the packet numbers you want to see by examining a brief report of all packets.

To select just one packet, you specify one packet number. For example, /PA:123.4 selects only packet number 123.4. To select a range of packets, you specify the first and last packet numbers of that range. For example, /PA:123.4:432.1 selects all the packets from packet 123.4 through packet 432.1.

MCR Arguments	DCL Arguments
mmm.mmm[:nnn.nnn]	mmm.mmm[:nnn.nnn]
:	ALL

You can specify more than one packet or packet range by using the special syntax described in Section 3.2.3.

The asterisk (*) indicates an open-ended number. You can select all the packets before a particular number (*:235.3), or all the packets after a particular number (235.3:*).

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /PACKET switch as follows:

```
/PACKET:*:*
```

3.3.1.4 The /SERIAL Switch

Format

```
/SE[RIAL]:argument
```

```
ANALYZE/ERROR_LOG/SERIAL_NUMBER:argument
```

The /SERIAL switch allows you to select packets based on their drive or pack serial number or both. This switch only applies to peripheral errors. You can select packets from any device that has a serial number by drive serial number, but you can only select packets from MSCP and last-track devices by pack serial number. Appendix B explains where RPT gets drive serial numbers and lists the significant digits in serial numbers for each error logging device.

The arguments for the /SERIAL switch allow you to specify one drive or one pack serial number, or both, in the same command line.

You can use the following arguments with the /SERIAL switch, where n is the drive or pack serial number:

MCR Arguments	DCL Arguments
D[RIVE]:n	DRIVE:n
P[ACK]:n	PACK:n
D[RIVE]:n,P[ACK]:n	DRIVE:n,PACK:n

3.3.1.5 The /TYPE Switch

Format

```
/T[YPE]:argument
```

```
ANALYZE/ERROR_LOG/INCLUDE:argument
```

The /TYPE switch selects error log packets based on their packet type. Use one or more of the following arguments with the /TYPE switch:

MCR Arguments	DCL Arguments
A[LL]	ALL
C[ONTROL]	CONTROL
E[RRORS]	ERRORS
M[EMORY]	MEMORY
PE[RIPHERAL]	PERIPHERAL
PR[OCESSOR]	PROCESSOR
S[YSTEM_INFORMATION]	SYSTEM_INFORMATION

The /TYPE switch arguments are described as follows:

ALL
Selects all error log packets in the error log file.

CONTROL
Selects error log command packets that control the Error Log Interface task (ELI).

ERRORS
Includes information on all peripheral, processor, and memory errors in the error log file.

MEMORY
Includes information from all events that occur in memory, such as memory parity errors.

PERIPHERAL
Includes information from all peripheral devices in the system that support error logging. This argument does not include logging of system information, such as mounts and dismounts, for the devices. (Use the SYSTEM_INFORMATION argument to display that information.)

PROCESSOR
Includes information from events that occur in the CPU itself, such as unknown interrupts.

SYSTEM_INFORMATION
Includes information from events that occur on the system, but are not specifically tied to a single piece of hardware. For example, if you use the SYSTEM_INFORMATION argument, error logging includes information on time changes, system service messages, and device-specific events (such as mounts and dismounts).

You can specify more than one type of packet by using the special syntax for the /TYPE switch, described in Section 3.2.3. However, note that the ALL argument and the ERRORS argument cannot be used with other arguments, but must be specified alone.

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /TYPE switch as follows:

```
/TYPE:ALL
```

3.3.1.6 The /VOLUME Switch

Format

```
/V[OLUME]:volume_label
```

```
ANALYZE/ERROR_LOG/VOLUME_LABEL:volume_label
```

The /VOLUME switch selects packets based on the specified volume label, as illustrated in the following example:

```
RPT>/T:PE/V:ERRLOGSYS RET
```

This command line specifies that RPT find the device or devices that contain a volume with the label ERRLOGSYS, and generate a report of peripheral errors on those devices. Since the /TYPE switch specification did not include system information, the report will not include mounts or dismounts for the devices.

3.3.2 The /FORMAT Switch

Format

```
/F[ORMAT]:argument

ANALYZE/ERROR_LOG/qualifier
```

The /FORMAT switch tells RPT how to format a report from packets in the error log file. You can select reports in brief format (one line for each error), in full format (all the information from the error log packets specified), or in register format (dumping only the registers for device errors).

Note that there is no single DCL command equivalent to the MCR /FORMAT switch. Instead, DCL uses qualifiers to the ANALYZE/ERROR_LOG command.

MCR Arguments	DCL Qualifiers
B[RIEF]	/BRIEF
F[ULL]	/FULL
R[EGISTER]	/REGISTERS
N[ONE]	/NODETAIL

The following sections describe the /FORMAT arguments in detail.

3.3.2.1 Brief Reports

Brief reports are short, one-line-per-packet reports on selected packets.

The brief report shown in Example 3–1 displays one line of information about each of the error log packets specified in the RPT command line. The annotated list following the example describes the sections in the brief report. The numbers in the list correspond to the numbers of the sections in Example 3–1 .

The following RPT command line generated the brief report in Example 3–1:

```
RPT>EXMBRIEF1.RPT=RAISIN.LOG/PA:(2.1,5.1,5.2,6.4,10.3,42.2) RET
```

Example 3-1: Error Log Brief Report

```
RSX-11M-PLUS Error Logging System V2.00  11-MAY-1987  11:05:18                    Page     1

 ❶          ❷                    ❸                ❹       ❺                              ❻
Entry      Time Stamp          Entry Type        Device  Error Type                     Additional Information
-------  --------------------  ----------------  ------  -----------------------------  ----------------------------------------

    2.1  01-MAY-1987 11:39:23  Device Hard Error  DL001:  Header not Found               Function = Write Data
    5.1  01-MAY-1987 12:43:47  Device Soft Error  DM004:  Data Late                       Function = Write Data

    5.2  01-MAY-1987 12:43:50  Device Soft Error  DM004:  Data Late                       Function = Write Data
    6.2  01-MAY-1987 12:44:02  Device Soft Error  DM004:  Data Late                       Function = Write Data
   10.3  01-MAY-1987 13:33:32  Device Soft Error  DM004:  Data Late                       Function = Read Data
   42.2  01-MAY-1987 15:26:46  Device Soft Error  DM003:  Data Late                       Function = Write Data

Selection information:
--------- -----------

❼   Command line: EXMBRIEF1.RPT=RAISIN.LOG/PA:(2.1,5.1,5.2,6.4,10.3,42.2)

❽   Report file: DR5:[BRADFORD]EXMBRIEF1.RPT;1
    Input file:  DR0:[1,6]RAISIN.LOG;1

❾   Report format selection:
    BRIEF
    WIDE

❿   Packet time selections:
    From * through *

⓫   Volume label selection:
    (not used)

⓬   Serial number selections:
    Drive:  (not used)
    Pack:   (not used)

⓭   Summary selections:
    No History
    No Error
    No Geometry
```

ZK-6308/1-HC

(Continued on next page)

Example 3-1 (Cont.): Error Log Brief Report

```
RSX-11M-PLUS Error Logging System V2.00  11-MAY-1987  11:05:18                    Page      2

(14)  Packet type selections:
         Processor
         Memory
         System_info
         Peripheral
         Control

(15)  Packet number selections:
         2.1
         5.1
         5.2
         6.4
         10.3
         42.2

(16)  Device selections:
         (ALL)

(17)  Number of packets printed / processed:
         6. / 6.

(18)  Processing began at 11-MAY-1987 11:04:23
      Processing ended at 11-MAY-1987 11:05:24
```

ZK-6308/2-HC

❶ The error log packet Entry Number, which describes the relative position in the error log file. This number does not change unless the file is changed; for example, by an ELI/APPEND command. It is not changed by normal logging into the file.

❷ The date and time the packet was logged.

❸ The type of entry in the error log file; for example, hard or soft device errors or system information.

❹ The device on which the error occurred.

❺ The error type as defined by the hardware information. RPT does not do any interpretation of these errors; it merely reports the hardware information.

❻ Any other information error logging gathers on the error, such as the I/O function that occurred at the time of the error.

❼ The RPT command line that generated the report.

❽ Input and report file specifications.

❾ The format selection.

❿ The packet time-range selection (the time the packets were created).

⓫ The volume label selection.

⓬ The drive and pack serial number selections.

⓭ The types of summary reports selected.

⓮ The packet type selections.

⓯ The packet number selections.

⓰ The device selections.

⓱ The number of packets printed and processed.

⓲ The time RPT report generation began and ended.

3.3.2.2 Full Reports

Full reports provide a detailed listing of device events. They list and interpret all of the information collected in the error log packets they specify.

The full report shown in Example 3–2 displays the complete contents of error log packet number 2.1, a header-not-found error described in the brief report in Example 3–1. The annotated list following the example describes the sections of the full report. The numbers on the list correspond to the numbered sections in Example 3–2.

The following RPT command line generated the full report in Example 3–2:

```
RPT>EXEMPF.RPT=RAISIN.LOG/PA:2.1/F:F [RET]
```

Example 3–2: Error Log Full Report

```
RSX-11M-PLUS Error Logging System V2.00  11-MAY-1987  16:15:17                                   Page      1

❶    Entry     2.1  Sequence 1.        DL001:  Device Hard Error     (Header not Found)    01-MAY-87 11:39:23

❷  System Identification:
   ------ ---------------

     System          Ident   Processor    Mapping   CPU   Format
     RSX-11M-PLUS    37      PDP-11/70    22-Bit    CPA   2.

❸  Device Identification Information:
   ------ -------------- ------------

     Device    Type          Volume Label    Controller    Unit    Subunit    Pack SN          Drive SN
     DL001:    RL02          <null label>    DL A          1       N/A        N/A              N/A
     I/O Count         Words Transferred         Cylinders Crossed     Hard Errors    Soft Errors
     2376.             12398848.                 2393/                 0.             0.

❹  I/O Operation Identification:
   --- --------- ---------------

     TI:      UIC         Task Name    Address      Length    Maximum Retries    Retries Remaining    Operation
     TT044:   [007,064]   BADT44       1630340      7680.     8.                 8.                   IO.WLB ! IQ.X

❺  Concurrent I/O Activity:
   ---------- --- ---------

     Device    Controller    Unit    Subunit    TI:      UIC         Task      Address      Length      Operation
     DR000:    DR L          0       N/A        C0000:   [031,076]   DR00F1    1212360      512.        IO.RLB
     MM000:    MM T          0       0          TT003:   [031,076]   BRUT3     N/A          N/A         IO.SPF

❻  I/O Operation Information:
   --- --------- ------------

     Device Function                Type of Error
     Write Data                     Header not Found
```

ZK-6309/1-HC

(Continued on next page)

Example 3-2 (Cont.): Error Log Full Report

```
RSX-11M-PLUS Error Logging System V2.00  11-MAY-1987  16:15:17                    Page      2

     Entry     2.1      (continued)

❼  Device Error Position Information:
   ------ ----- -------- ------------

     Cylinder   Group   Head   Sector    Block

     349.       N/A     0.     40.       13980.

❽  Device Supplied Information:
   ------ -------- ------------

     Name     Value       Interpretation

     RLCS     112733      *[15   ] Composite Error                    *[13:10] Error Code = Header not Found
                           [ 9: 8] Drive Selected = 1                   [ 7   ] Controller Ready
                           [ 6   ] Interrupt Enabled                    [ 5: 4] BA17,BA16 = 01 (B)
                           [ 3: 1] Function = Write Data                [ 0   ] Drive Ready

     RLBA     017000       [15: 0] Bus Address Register

     RLDA     12725C       [15: 7] Cylinder Address = 349.              [ 6   ] Head Selected = Upper Head
                           [ 5: 0] Sector Address = 40.

     RLMP1    133061       [15: 0] Register contents undefined

     RLMP1    000235       [ 7   ] Drive Type = RL02                    [ 6   ] Head Address = Upper Head
                           [ 4   ] Heads Out (over the disk)            [ 3   ] Brushes Home
                           [ 2: 0] Drive State = Lock On
```

ZK-6309/2-HC

❶ The same identification information listed in items 1 through 5 of the brief report description.

❷ System identification information including operating system and base level, CPU type, and address mapping type.

❸ Device identification information including the device name, device type, volume label, controller, unit number, pack and drive serial numbers, total I/O count on the device, the amount of data transferred in decimal words, the number of cylinders crossed while accessing data, and the number of hard and soft errors logged previous to this device.

❹ I/O operation identification includes the terminal and UIC that initiated the operation, the task name, the beginning physical memory address of the I/O buffer, the length of the I/O request (in bytes), the maximum number of retries the device driver allows for an I/O operation, the number of retries remaining, and the actual I/O operation taking place.

❺ Concurrent I/O activity occurring on other devices when this error occurred. This section is present only when concurrent I/O activity takes place; otherwise, the section is suppressed.

❻ I/O operation information includes the device I/O function and type of error as defined by the hardware.

❼ The device error position information locates the error by cylinder, group, head, sector, and logical block number.

❽ The device-supplied information includes a dump of the device registers according to name, contents, and interpretation of the bits in the registers. The asterisk (*) beside some bit interpretations means that the condition is likely to have contributed to the error. It is a sign that you may want to examine the condition.

3.3.2.3 Register Reports

Register reports contain the same information as full reports for all events except those that occur on peripherals. Register reports list the contents of all device registers for peripherals, but contain no other information.

The register report in Example 3–3 includes only the register section of the full report (see Example 3–2) for packet 2.1 (the header-not-found error).

The following RPT command line generated the register report shown in Example 3–3:

```
RPT>EXEMPN.RPT=RAISIN.LOG/F:R/PA:2.1 [RET]
```

3.3.2.4 No Report

RPT does not generate a formatted output report on event information. The NONE argument causes RPT not to format the packets or produce a packet-by-packet report. The NONE argument is useful when you want to generate only a summary report.

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /FORMAT switch as follows:

```
/FORMAT:BRIEF
```

Example 3–3: Error Log Register Report

```
RSX-11M-PLUS Error Logging System V2.00  11-MAY-1987  16:15:43                                                    Page      1

   Entry     2.1  Sequence 1.        DL001:  Device Hard Error     (Header not Found)     01-MAY-87 11:39:23

  Device Supplied Information:
  ------ -------- ------------

    Name      Value       Interpretation

    RLCS      112733      *[15   ] Composite Error                          *[13:10] Error Code = Header not Found
                           [ 9: 8] Drive Selected = 1                        [ 7   ] Controller Ready
                           [ 6   ] Interrupt Enabled                         [ 5: 4] BA17,BA16 = 01 (B)
                           [ 3: 1] Function = Write Data                     [ 0   ] Drive Ready

    RLBA      017000       [15: 0] Bus Address Register

    RLDA      12725C       [15: 7] Cylinder Address = 349.                   [ 6   ] Head Selected = Upper Head
                           [ 5: 0] Sector Address = 40.

    RLMP1     133061       [15: 0] Register contents undefined

    RLMP1     000235       [ 7   ] Drive Type = RL02                         [ 6   ] Head Address = Upper Head
                           [ 4   ] Heads Out (over the disk)                 [ 3   ] Brushes Home
                           [ 2: 0] Drive State = Lock On
```

ZK-6310-HC

3.3.3 The /SUMMARY Switch

Format

```
/SU[MMARY]:argument[s]

ANALYZE/ERROR_LOG/STATISTICS:argument[s]
```

The /SUMMARY switch tells RPT how to summarize the information from packets in the error log file. Since the summaries are compilations of the data gathered from the individual packets, the /SUMMARY switch tells RPT what particular piece of information from the packets to use as the basis for a summary report.

RPT cannot create summary reports in narrow width. If you specify narrow width by using the /W:N switch, RPT formats the packet-by-packet display in narrow format, but formats the summary in wide format.

You can use the following arguments with the /SUMMARY switch to generate summary reports:

MCR Arguments	DCL Arguments
ALL	ALL
ERROR	ERROR
GEOMETRY	DISK_GEOMETRY
HISTORY	HISTORY
NONE	NONE

To generate more than one summary report from the same command line, use the special syntax described in Section 3.2.3. For example, the following command produces an ERROR report and a GEOMETRY report:

```
RPT>=/SU:(ERROR,GEOMETRY)
```

Note, however, that the ALL and NONE arguments may not be specified with other arguments.

The /SUMMARY switch arguments are described in the following sections.

3.3.3.1 The ALL Argument

RPT generates summary reports sorted by error, geometry, and history. These summary reports are described in Sections 3.3.3.2 through 3.3.3.4.

3.3.3.2 The ERROR Argument

RPT generates a summary report sorted by error type. The ERROR summary, sorted by device, shows the number of times each error occurred on the device. The Count column of the summary tells the number of times the error occurred. Example 3–4 shows the summary section of an ERROR summary report.

The following RPT command generated the report in Example 3–4:

```
>RPT ERRORRPT.LOG=/SU:E/F:N RET
```

When you specify /FORMAT:NONE, RPT does not display packets on a packet-by-packet basis as shown in the previous examples.

Example 3-4: ERROR Summary Report

RSX-11M-PLUS Error Logging System V2.00 01-MAY-1987 16:16:16 Page 1

Error Summary:

Device	Type	Drive SN	Volume Label	Pack SN	Error Type	Count	First/Last Occurence	Entry
DM003:	RK07	510	<null label>	N/A	Data Late	20.	30-APR-1987 10:34:42	5.1
							30-APR-1987 16:15:14	55.1
					Error Correction Hard	2.	30-APR-1987 13:02:12	30.4
							30-APR-1987 13:02:13	31.1
					No error bit found	31.	30-APR-1987 14:52:31	32.2
							30-APR-1987 15:20:47	45.1
			MAINDEV	3017B	No error bit found	9.	30-APR-1987 12:36:04	26.1
							30-APR-1987 15:21:18	46.3
					Loading Heads	1.	30-APR-1987 15:20:53	45.3
							30-APR-1987 15:20:53	45.3
					Data Late	11.	30-APR-1987 15:34:19	50.3
							30-APR-1987 16:33:45	58.3
DM004:	RK07	86	<null label>	N/A	Error Correction Hard	6.	30-APR-1987 11:04:24	10.4
							30-APR-1987 11:04:28	12.2
					Data Late	42.	30-APR-1987 11:06:13	13.1
							30-APR-1987 15:31:44	49.3
					Data Check	1.	30-APR-1987 11:10:10	24.2
							30-APR-1987 11:10:10	24.2
			SYSTEM	72237	Data Late	1.	30-APR-1987 16:36:26	59.1
							30-APR-1987 16:36:26	59.1
			USER	2E7247	Data Late	5.	30-APR-1987 10:39:22	7.1
							30-APR-1987 10:39:28	8.2
DR002:	RM03	4352	TESTDEV	8032C	Write Lock Error	1.	30-APR-1987 12:29:05	25.2
							30-APR-1987 12:29:05	25.2
DR003:	RM03	8143	<null label>	N/A	Invalid Address Error	1.	01-MAY-1987 09:32:33	104.1
							01-MAY-1987 09:32:33	104.1

ZK-6311-HC

3.3.3.3 The GEOMETRY Argument

RPT generates a summary report based on device geometry (logical block or sector, for example). The error count column of the summary tells how many times an error occurred in that device location.

The following RPT command generated the report in Example 3–5:

```
>RPT ERRORRPT.LOG=/SU:G/F:N [RET]
```

3.3.3.4 The HISTORY Argument

RPT generates a summary report sorted by device error history. It displays the hard and soft error count and QIO count for every volume used on each device.

The following RPT command line generated the report in Example 3–6:

```
>RPT ERRORRPT.LOG=/SU:H/F:N [RET]
```

3.3.3.5 The NONE Argument

The NONE argument is invoked when you use the RPT default command line. It causes RPT to generate no summary report. RPT invokes this argument to satisfy the requirement that the command line specify how to summarize the information from error log packets.

3.3.4 The /REPORT Switch

Format

```
/R[EPORT]:defined_report_string

ANALYZE/ERROR_LOG/COMMAND=defined_report_string
```

The /REPORT switch invokes a predefined string of switches for RPT to use. The switch string usually defines a particular type of report, such as a report for a particular time period. The string contains any valid combination of RPT switches. The string cannot include the /REPORT switch.

The /REPORT switch allows you to access a file that contains the switch combinations you use frequently and lets you invoke the switches by using the string name, instead of reentering the switches explicitly.

RPT uses the normal default values described in Section 3.2.1 for all switches not defined in the switch string, if the switches have defaults.

The DIGITAL and user-defined switch strings are found in a control file module, PARSEM, or in a disk file, LB:[1,6]ERRDEFINE.CFS, respectively. The /REPORT switch first searches PARSEM, where it finds DIGITAL-defined strings. If the string is not defined there, RPT searches ERRDEFINE.CFS.

Since RPT looks in the control file module first, you cannot redefine the DIGITAL-supplied strings unless you alter the control file module. DIGITAL does not recommend that you alter control file modules. You can change the definitions for DIGITAL-supplied strings by slightly altering their names and inserting the switch under the new name in ERRDEFINE.CFS.

Example 3-5: GEOMETRY Summary Report

RSX-11M-PLUS Error Logging System V2.00 01-MAY-1987 16:19:57 Page 1

Geometry Summary:

Device	Type	Drive SN	Volume Label	Pack SN	Head	Group	Cylinder	Sector	LBN	Error Count
DM003:	RK07	510	<null label>	N/A	0.	N/A	0.	0.	0.	1.
					0.	N/A	3.	20.	218.	1.
					0.	N/A	12.	2.	794.	1.
					0.	N/A	13.	9.	867.	1.
					0.	N/A	41.	0.	2706.	1.
					0.	N/A	52.	0.	3432.	2.
					1.	N/A	16.	5.	1083.	1.
					1.	N/A	27.	14.	1818.	1.
					1.	N/A	30.	11.	2013.	1.
					1.	N/A	137.	21.	9085.	1.
					2.	N/A	17.	5.	1171.	1.
					2.	N/A	35.	16.	2370.	1.
					2.	N/A	42.	1.	2817.	1.
					2.	N/A	235.	12.	15566.	1.
					2.	N/A	297.	13.	19659.	1.
					2.	N/A	423.	2.	27964.	1.
			SYSTEM	72237	0.	N/A	495.	3.	32673.	1.
			USER	2E7247	2.	N/A	598.	10.	39522.	2.
					2.	N/A	598.	9.	39521.	1.
					2.	N/A	598.	8.	39520.	2.
DR002:	RM03	4352	TESTDEV	8032C	4.	N/A	322.	31.	51678.	1.
DR003:	RM03	8143	<null label>	N/A	0.	N/A	0.	30.	29.	1.

ZK-6312-HC

Example 3–6: HISTORY Summary Report

```
RSX-11M-PLUS Error Logging System V2.00  01-MAY-1987 16:24:09                    Page      1

History Summary:

Device      Type         Volume Label    Pack SN      Total QIOs    Hard Errors  Soft Errors
------      ------------ ------------    -----------  -------------  -----------  -----------

DB002:      RP04/05      WORKDEV         N/A          364.

DL000:      RL02                         N/A          8143.
                         GENERIC         3499         609.

DL001:      RL02                         N/A          8439.
                         CABRO           3039         10.

DM003:      RK07                         N/A          54232.         2.           50.
                         MAINDEV         3017B        15613.         1.           21.

DM004:      RK07                         N/A          24782.         41.          8.
                         UPDATES         2373E        6.
                         SYSTEM          ADE3         1492.
                                         72237        2125.                       1.
                         USER            2E7247       1702.                       5.

DR002:      RM02/03      TESTDEV         8032C        10.

DR003:      RM02/03      IODEV           16AF2        4.                          1.

EM000:      ML11                         N/A          1191.
                         WORKMAINT       N/A          3.

MM001:      TE/U16/45/77                 N/A          4631.                       6.

MM002:      TE/U16/45/77                 N/A          77808.                      21.
```

ZK-6313-HC

3.3.4.1 Predefined Switch Strings

DIGITAL supplies the following four predefined switch strings to use with the /REPORT switch:

Predefined Switch String	Switches Defined
DAY	/FO:FULL/SU:ALL/DA:TODAY
WEEK	/SU:(HISTORY,ERROR)/DA:PREVIOUS:7
MONTH	/SU:(HISTORY,ERROR)/DA:PREVIOUS:31
SYSTEM	/SU:(HISTORY,ERROR)

Note that the names of the predefined switch strings must be entered in full. They cannot be abbreviated.

3.3.4.2 User-Defined Switch Strings

You can name and define your own switch strings to use with the /REPORT switch by creating and editing LB:[1,6]ERRDEFINE.CFS and inserting the switch strings you want to define.

Entries in this file must be in the following form:

```
'switchname','switchstring'
```

switchname

The name of the switch string you are defining. This name becomes the argument to the /REPORT switch when you want to invoke the string. (The name must be nine or fewer characters in length.)

switchstring

The RPT switches you select to generate the report. (The switch string must be 80 or fewer characters in length.)

Note

The single quotation marks are a required part of the syntax. You also must enter the full switch string name when you invoke a user-defined string.

For example, if you want to generate a brief report of peripheral errors on all the DB devices on your system, edit ERRDEFINE.CFS and insert the following line:

```
'DB','/FO:B/TY:PE/DE:DB'
```

You can then create this report with the following RPT command:

```
RPT> outfile=infile/R:DB [RET]
```

3.3.5 The /WIDTH Switch

Format

```
/W[IDTH]:argument
```

```
ANALYZE/ERROR_LOG/qualifier
```

The /WIDTH switch allows you to set the line width of the report RPT generates to narrow (80 columns) or wide (132 columns). The basic report format does not change when RPT creates a narrow report. Instead, each long line of the report wraps onto the next line at an appropriate place.

Summary reports do not honor the /WIDTH switch. The summary portion of these reports is always in the wide format.

Note that there is no DCL command equivalent to the MCR /WIDTH switch. Instead, DCL uses qualifiers to the ANALYZE/ERROR_LOG command.

You can use the following arguments with the /WIDTH switch:

MCR Arguments	DCL Qualifiers
N[ARROW]	/NOWIDE
W[IDE]	/WIDE

When you use the default RPT command line (see Section 3.2.2), RPT invokes the /WIDTH switch as follows:

```
/WIDTH:WIDE
```

3.4 ERLCNF Report Messages

The error log control file displays messages on your terminal if errors occur during report generation. The messages include an abbreviation, a severity level code for the error (warning, informational, or fatal), and text describing the error.

In some cases, RPT also writes the message in the error log report, if it explains an error that appears in the report. For example, when RPT fails to find a control file module for a device you specify, it displays a message on your terminal and in the report that includes the error message.

The following sections list the ERLCNF messages, along with possible causes and methods for recovery.

3.4.1 ERLCNF Fatal Errors

ERLCNF-F-ARGNOTUNQ, Argument specification <argument> is not unique

Explanation: You did not specify enough characters in a switch argument to make it unique. It will be confused with another argument.

User Action: Check the argument syntax and reenter the command.

ERLCNF-W-BADSUBPKT, Possible corruption in the <packetname> subpacket in item <item label>

Explanation: RPT found something in the subpacket that appeared to be abnormal. The file may be corrupted or it may be an internal error within RPT.

User Action: You should never see this message. If you do, send in a Software Performance Report (SPR) along with a dump of the packet that generated the message and any other information you have. You can create a dump of the packet using the starting virtual block number of the packet (the nnn portion of the packet number nnn.m).

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-W-DUILLFORM, MSCP format code <code> is undefined

Explanation: This may be an internal error within RPT. It indicates a format code in the RA80 packet that is corrupted.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

ERLCNF-F-ILLARGCOM, Illegal argument combination

Explanation: You specified an illegal combination of arguments with a switch.

User Action: Check the syntax and reenter the command.

ERLCNF-F-ILLFILSPC, Illegal file specification - <filename>

Explanation: You used an illegal file specification with an RPT report-generating command.

User Action: Check the syntax and try the operation again.

ERLCNF-W-ILLPACCOD, Illegal code in packet <packetid> , Code = <xx>

Explanation: The major code for the indicated packet is beyond the range that RPT can handle.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-F-ILLPACRAN, Illegal packet range - LOW = <xx> , HIGH = <xx>

Explanation: The RPT Packet Selection (/PACKET) switch requires arguments to be packet numbers in a specific format.

User Action: Determine the correct number for the packet you want to display, check the syntax, and reenter the command.

ERLCNF-W-ILLPACSBC, Illegal subcode in packet <packetid> , Code = <xx> , Subcode = <xx>

Explanation: The subcode for the indicated packet is beyond the range that RPT can handle.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-F-ILLPACSPC, Illegal packet specification <packetid>

Explanation: You used an invalid packet specification with an RPT report-generating command.

User Action: Resubmit the command file with the correct packet specification.

ERLCNF-F-ILLSWTARG, Illegal switch argument - <argument>

Explanation: RPT recognized the switch argument, but determined that the argument is incorrect in the context given.

User Action: Check the syntax and reenter the command.

ERLCNF-F-INTERR001, Internal error detected at position number <n>

Explanation: This is an internal RPT error. It occurs with the PARSECLST and PARSECTION error messages.

User Action: You should never see this message. If you do, send in an SPR and the command line that generated the message and any other information you have.

ERLCNF-F-MODNOTFND, Module not found - <module>

Explanation: RPT searched ERRLOG.ULB for the module and did not find it.

User Action: You should never see this message. If you do, send in an SPR and the command line that generated the message. Be sure to include the name of the module that was missing.

ERLCNF-F-MULARGSPC, Argument <argument> specified multiple times

Explanation: You specified an RPT switch argument more than once.

User Action: Check the syntax and reenter the command.

ERLCNF-F-MULSWTSPC, Switch <switch> specified multiple times

Explanation: You entered the specified switch more than once on the same RPT command line. RPT allows you to specify each switch only once.

User Action: Check the syntax and reenter the command. Use the special syntax for multiple switch specifications described in Section 3.2.3, if the switch allows it.

ERLCNF-F-NODIDPACK, No DEVICE_ID subpacket

Explanation: This is an internal error within RPT.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have.

ERLCNF-W-NODRIVSZ, No drive of size <size> for mnemonic <ddnn> ; using EUNKWN

Explanation: This may be an internal error within RPT.

User Action: You should never see this message if you have only DIGITAL hardware. If you have non-DIGITAL hardware and you receive this message, it is caused by a disagreement between RPT's table of device sizes and the actual size of the device. See Section 4.4.2 for information on changing the table of device sizes.

ERLCNF-W-NODRIVTYP, No drive type <type> for mnemonic <dd> using EUNKWN

Explanation: This may be an internal error within RPT. From the mnemonic, the drive appears to be a MASSBUS device. However, RPT does not recognize the device type as a MASSBUS device.

User Action: You should never see this message if you have only DIGITAL hardware. If you have non-DIGITAL hardware, the error is caused by a disagreement between RPT's table of device sizes and the size of the actual device. See Section 4.4.2 for information on changing the table of device sizes.

ERLCNF-F-NOINPFILE, No input file specified

Explanation: RPT did not find an input file on the command line. This message occurs when you fail to specify an equal sign (=) in the command.

User Action: Check the syntax and reenter the command.

ERLCNF-W-NONOTES, No notes available for device <devicename>

Explanation: RPT includes a facility for displaying notes at the bottom of Full or Register reports. This internal error message indicates that a device did not have an associated NOTES module.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-F-NOREMATCH, No predefined switch string for <string>

Explanation: RPT did not find the defined report string you used in a /REPORT command, either in ERRDEFINE.CFS or among the DIGITAL-defined report strings. Remember to use the entire name of the DIGITAL or user-defined string.

User Action: Check the syntax and reenter the command.

ERLCNF-F-OPNINPFIL, Failed to open the input file

Explanation: RPT could not open the input file specified. This message is accompanied by the FILERRCOD information message, which displays the FCS error code from the file.

User Action: Check the FCS error code and retry the command after correcting the indicated condition.

ERLCNF-F-OPNREPFIL, Failed to open the report file

Explanation: RPT could not open the report (output) file specified. This message is accompanied by the FILERRCOD information message, which displays the FCS error code from the report file.

User Action: Check the FCS error code and retry the command after correcting the indicated condition.

ERLCNF-F-OPNUSRFIL, Failed to open the user file

Explanation: RPT could not open the user file specified. This message is accompanied by the FILERRCOD information message, which displays the FCS error code from the file.

User Action: Check the FCS error code and retry the command after correcting the indicated condition.

ERLCNF-F-SWTNOTUNQ, Switch specification <switch> is not unique

Explanation: You did not specify enough characters of a switch to make it unique. It will be confused with another switch.

User Action: Check the switch syntax and reenter the command.

ERLCNF-F-SYNTAXERR, Command line syntax error

Explanation: Some element of the command line does not have the correct syntax.

User Action: Check the syntax and reenter the command.

ERLCNF-F-TOOFEWARG, Too few arguments in switch <switchname>

Explanation: You specified a switch that requires one or more arguments without specifying enough arguments.

User Action: Check the syntax and reenter the command.

ERLCNF-F-UNKNWARG, Unknown argument - <argument>

Explanation: You specified an argument that is unknown to RPT.

User Action: Check the syntax and reenter the command.

ERLCNF-W-UNKNWNDEV, Device mnemonic <dd> is unknown; using EUNKWN

Explanation: This may be an internal error within RPT.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-W-UNKNWNNOT, No note number <number> for device <devicename>

Explanation: RPT includes a facility for displaying notes at the bottom of reports. This internal error message indicates that a device tried to print a note that was not available.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-F-UNKNWSWT, Unknown switch - <switchname>

Explanation: You specified an unknown RPT switch.

User Action: Check the syntax and reenter the command.

3.4.2 ERLCNF Warning Message

ERLCNF-W-USEEUNKWN, Module <modulename> not found; using EUNKWN

Explanation: RPT was not able to find the module specified in the error logging universal library and went to the EUNKWN module instead. This causes a formatted dump of the device register to appear in the report. This message usually occurs if you tune your universal library and eliminate the module for a device you want to use.

User Action: Retune the universal library to include the missing module.

3.4.3 ERLCNF Informational Messages

These messages accompany other ERLCNF messages to give you additional information. They do not affect RPT operation.

ERLCNF-I-FILERRCOD, File error code = <errorcode>

Explanation: This message displays the FCS error code for a file. It accompanies messages on file-access failures.

User Action: None is necessary. This is an informational message.

ERLCNF-I-PARSECLST, PARSE.SECTION_LIST = <buf>

Explanation: This is an internal error within RPT. This message accompanies the INTERR001 message described previously.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

ERLCNF-I-PARSECTION, PARSE.SECTION = <buf>

Explanation: This is an internal error within RPT. It accompanies the INTERR001 message described previously.

User Action: You should never see this message. If you do, send in an SPR along with a dump of the packet that generated the message and any other information you have. (See the BADSUBPKT description.)

If the message refers to a packet that you have altered or a module that you wrote, correct the module, recompile, and add it to the library.

3.5 ERLRPT Report Messages

Most of the following error messages are either associated with errors in the control file module that RPT is interpreting or are internal RPT errors.

If the message refers to a control file module that you have altered or a module that you wrote and added to the error logging system, correct the error, recompile the module, and add it to the library. The module in which the error occurred is specified in the first (or top) line of the execution stack dump produced by RPT. This information appears on the report file and on the terminal from which RPT is being run.

If the message refers to a DIGITAL-supplied module or is an internal RPT error, please submit a Software Performance Report (SPR) and include a listing of the error log report file produced by RPT.

ERLRPT-F-ACCUDFVAR, Attempt to access undefined variable.

Explanation: A control file module attempted to access a variable that had not been defined.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-BADDIGIT, Invalid numeric digit in conversion.

Explanation: A numeric literal or the ASCII string argument for the %COD$OCTAL, %COD$DECIMAL, %COD$HEX, %COD$BCD, %COD$BINARY, or %COD$MACHINE function contained an invalid character for the specified radix or was null or blank.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-BITFLDSIZ, Bit or field too large in extraction operation.

Explanation: The bit or field in an extraction operation exceeded the size of the value on which the extraction was performed.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-BITTOOHIG, Bit number too large for specified storage unit.

Explanation: The bit number specified by the character string portion of a #BI, #WI, #LI, #QI, or #VI numeric literal was too large for the specified value size.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-CASENOMAT, CASE selection expression has no matching value.

Explanation: No match was found for the value of the selector expression in a CASE statement, and no ELSE clause was specified in the CASE statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-CONTROLFI, Could not open control file.

Explanation: The control file module could not be opened.

User Action: If you are using the default control file library, check to see that it is in either LX:[1,6] or LB:[1,6] and is not locked, and that you have read access to it. If you are using a user-specified control file, check to see that it is not locked and that you have read access.

ERLRPT-F-COROUMIS, COROUTINE statement executed with no COROUTINE stack frame.

Explanation: A COROUTINE statement was executed without specifying a coroutine in the corresponding CALL statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-CRASH, Control file requested abort.

Explanation: The CRASH statement was executed by a control file module.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DATNOTEXI, Data declaration is longer than data.

Explanation: The amount of data specified in a PACKET or SUBPACKET declaration was larger than the amount of data in the packet or subpacket. This condition may be due to an error in the control file module or an error in the error log packet being analyzed.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DECAGAIN, Group in declaration already declared. Redeclaration illegal.

Explanation: A DECLARE, PACKET, SUBPACKET, TABLE, or DYNAMIC_TABLE statement was executed with a group name that was already defined.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DECGRPCTX, Group in DECODE statement has no context.

Explanation: The group in the DECODE statement was a TABLE, DYNAMIC_TABLE, or PACKET or SUBPACKET with the REPEATED attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DECNOBIT, No BIT declaration corresponding to DECODE list item.

Explanation: The bit number specified for a data item in the DECODE statement had no corresponding BIT declaration for the data item in the specified group.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DECNOTEXT, No bit to text translation for DECODE list item.

Explanation: The BIT declaration corresponding to the bit number specified for a data item in the DECODE statement had no print expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DEFCASELS, No match for control expression in CASE conditional definition.

Explanation: No match was found for the value of the selector expression in a CASE conditional definition and no ELSE clause was specified.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DEFNOCONT, Attempt to access data in variable in group with null context.

Explanation: The control file module attempted to access a variable in a TABLE, DYNAMIC_TABLE, or PACKET or SUBPACKET with the REPEATED attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-DEFNOSTAK, Declaration stack overflow.

Explanation: The stack used for processing declarations has overflowed.

User Action: Edit RPTBLD.CMD to increase the extension for program section DCSTK0, and rebuild RPT.

ERLRPT-F-DEFSTKUND, Internal error - Declaration stack underflow.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-DIVZERO, Attempt to divide by zero.

Explanation: A control file module attempted to divide by zero.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-EXEINVCOD, Internal error - Execution stack entry has invalid code.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-EXEINVPOS, Internal error - Input file has invalid position value.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-EXPGRPNOC, Attempt to reference POINTER for group without context.

Explanation: A control file module attempted to reference the POINTER special variable for a TABLE, DYNAMIC_TABLE, or PACKET or SUBPACKET with the REPEATED attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-EXPINVCOD, Internal error - Invalid expression item code in expression.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-EXPINVTYP, Internal error - Invalid symbol data type in expression.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-EXPNORSYM, Symbol without read access referenced in expression.

Explanation: A control file module attempted to read a variable defined in a DECLARE statement that had not been initialized.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-EXPUDFGRP, Undefined group referenced in expression.

Explanation: A control file module attempted to reference a group that had not been defined.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-EXPUDFSYM, Undefined symbol referenced in expression evaluation.

Explanation: A control file module attempted to access an undefined symbol.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-EXPVALOVR, Value stack overflow during expression evaluation.

Explanation: The stack used for processing values and expressions has overflowed.

User Action: Edit RPTBLD.CMD to increase the extension for program section VLSTK0, and rebuild RPT.

ERLRPT-F-EXPVARNOC, Attempt to access variable without context in expression.

Explanation: A control file module attempted to reference a variable in a TABLE, DYNAMIC_ TABLE, or PACKET or SUBPACKET with the REPEATED attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FILERCLOS, File close error.

Explanation: An error occurred when RPT attempted to close a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLRPT-F-FILERREAD, File read error.

Explanation: An error occurred when RPT attempted to read a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLRPT-F-FILERSPAN, Records in file are not allowed to span blocks.

Explanation: The span block attribute of the error log file being analyzed was set. ELI creates the error log file with this attribute set, and neither ELI, ERRLOG, nor RPT will modify it, but some other task may have.

User Action: Use ELI to start or restart error logging with a new version of the error log file, then use the Peripheral Interchange Program (PIP) to append the previous version to the new version. PIP may produce the following warning message:

```
PIP -- Input files have conflicting attributes
```

This message can be ignored.

ERLRPT-F-FILERWRIT, File write error.

Explanation: An error occurred when RPT attempted to write to a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLRPT-F-FILINTOPN, Internal error - File already open.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-FILINVCOD, Internal error - invalid file code for specified operation.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-FILINVMOD, Control file library has invalid module name table format.

Explanation: The control file library has an invalid module name table format. The control file must be a universal library.

User Action: Ensure that the control file is a valid universal library and rerun RPT.

ERLRPT-F-FILNOTCTX, Operation requires that dynamic file have context.

Explanation: A control file module executed a POINTER DELETE or POINTER MOVE statement on a DYNAMIC_TABLE for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FILNOTEXI, Internal error - Declared dynamic file does not exist.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-FILNOTVIR, Could not create virtual address space for module table.

Explanation: RPT could not dynamically extend its address space to create room for the module table.

User Action: If the maximum task size for the partition is less than 32K words, use the MCR command SET /MAXEXT or DCL command SET SYSTEM/EXTENSION_LIMIT to increase the maximum task size, or run RPT in a different partition.

ERLRPT-F-FILTOOBIG, File too large to read.

Explanation: RPT cannot analyze error log files that are larger than $65{,}535_{10}$ blocks.

User Action: Use ELI to create new error log files more often.

ERLRPT-F-FINDFIELD, FIELD in FIND statement does not have valid data type.

Explanation: A control file module executed a FIND statement where the specified FIELD was not NUMERIC, STRING, ASCII, RSXTIME, VMSTIME, or LOGICAL.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FINDNOCON, FIND statement not valid on a group with no context.

Explanation: A control file module executed a FIND statement for a TABLE or DYNAMIC_TABLE attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORCLSNUL, FORMAT clause null.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a null FORMAT clause.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORFIELDS, FORMAT error - Field too narrow for variable to print.

Explanation: A control file module executed a WRITE_GROUP statement where the width specified by a !DP directive was too short for the corresponding variable.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORFIELDW, FORMAT error - Name too long for field in !DF directive.

Explanation: A control file module executed a WRITE_GROUP statement where the width specified in a !DF directive was less than the length of the name of the corresponding variable.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORINVCHA, FORMAT error - Invalid character in FORMAT clause.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a FORMAT clause that contains a nonprinting character.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORINVCHR, FORMAT error - Invalid character in string in !DP directive.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a !DP directive that contains an invalid character.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORINVDIR, FORMAT error - Invalid format directive code.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a FORMAT clause that contains an invalid format directive.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORINVVTY, FORMAT error - Attempt to output invalid variable type.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a FORMAT clause that contains a !DP directive for which the corresponding variable is the wrong type.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORLINEOV, FORMAT error - Line overflow in FORMAT clause.

Explanation: A control module executed a WRITE or WRITE_GROUP statement during which the output buffer overflowed while processing the FORMAT clause. The output buffer is 132 characters wide.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORNOARG, FORMAT error - Format directive missing required argument.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a FORMAT clause that contains an !FC or !FS directive with no numeric argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORNONAME, FORMAT error - Request to print a field name for a value.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a FORMAT clause that contains a !DF directive matched with a value rather than a variable.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORNOREAD, FORMAT error - Attempt to print a variable without read access.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement that attempted to print a variable without read access.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FORNOTASC, FORMAT clause not ASCII.

Explanation: A control file module executed a WRITE or WRITE_GROUP statement with a non-ASCII FORMAT clause.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNDATNOT, Specified (sub)packet is not large enough for offset.

Explanation: A control file module executed a look-ahead function where the value of the offset argument was larger than the specified packet or subpacket.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNFIELDS, Invalid conversion code argument to time conversion function.

Explanation: A control file module executed a time conversion function with an invalid value for the conversion code argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNINVPOI, Invalid string pointer value in string function.

Explanation: A control file module executed a %STR$PARSE or %STR$QUOTE function where the value of the pointer argument was larger than the length of the string argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNNOTCHA, Argument to %STR$CHAR is not in valid range for character.

Explanation: The value of the argument for the %STR$CHAR function must be in the range 0 to 127_{10}.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNNOTIMP, Function not implemented.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-FUNQUOODD, Quote string in %STR$QUOTE function must have even length.

Explanation: A control file module executed a %STR$QUOTE function, where the quote string argument was not an even length.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNSTRSIZ, Output string from string function too large.

Explanation: A control file module executed a string function that resulted in a string longer than 255 characters.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-FUNWRONGA, Incorrect number of arguments in function call.

Explanation: A control file module executed a function call with the wrong number of arguments.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-GROUPDEF, Attempt to reference undefined group.

Explanation: A control file module attempted to reference an undefined group.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-GROUPNOC, POINTER statement executed on a group without context.

Explanation: A control file module executed a POINTER statement on a TABLE or DYNAMIC_TABLE for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-HEAPOVERF, Heap too small to hold value. Overflow.

Explanation: The heap used for processing values and expressions has overflowed.

User Action: Edit RPTBLD.CMD to increase the extension for program section VHEAP0, and rebuild RPT.

ERLRPT-F-INCFORWRI, Too few FORMAT expressions in WRITE_GROUP statement.

Explanation: A control file module executed a WRITE_GROUP statement that did not have two FORMAT expressions in the FORMAT clause.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-INCRDECRL, Numeric variable in INCREMENT or DECREMENT larger than value.

Explanation: A control file module executed an INCREMENT or DECREMENT statement on a variable that was larger than a word.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-INCRDECRN, Variable in INCREMENT or DECREMENT statement not numeric.

Explanation: A control file module executed an INCREMENT or DECREMENT statement on a nonnumeric variable.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-INCRDECRV, Variable in INCREMENT or DECREMENT not valid or read-only.

Explanation: A control file module executed an INCREMENT or DECREMENT statement on a variable that was not both readable and writable.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-INTINVDEC, Internal error - Invalid declaration entry type in WRITEGROUP.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-INTVALSTK, Internal error - Statement left information on value stack.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-INVRADCNV, Internal error - invalid radix code for conversion.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-LEAVENOC, LEAVE statement executed outside of a conditional block.

Explanation: A control file module executed a LEAVE statement that was not inside a loop statement block.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-LISTNOEXP, No expression in LIST for corresponding SEARCH variable.

Explanation: A control file module executed a SEARCH statement in which a match was found, but there were not enough expressions in the list element for the number of variables specified in the GET clause of the SEARCH statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-LISTNOMAT, Too many expressions in SEARCH statement for referenced LIST.

Explanation: A control file module executed a SEARCH statement in which there were too many search expressions for the specified LIST.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-LISTNOTDF, Group referenced in SEARCH statement is not defined.

Explanation: A control file module executed a SEARCH statement in which the name specified for the LIST was not defined.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-LISTNOTLS, Group referenced in SEARCH statement is not a LIST.

Explanation: A control file module executed a SEARCH statement in which the name specified for the LIST was not defined as a list.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MATDIFTYP, Values of differing type cannot be matched.

Explanation: A control file module executed a MATCH statement that tried to match values of differing types.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MATVALSIZ, Values of different size cannot be matched.

Explanation: A control file module executed a MATCH statement that tried to match values of differing sizes.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MEMALLFAI, Memory allocation failure - insufficient virtual memory.

Explanation: RPT could not dynamically extend its address space to create room for DYNAMIC_TABLEs or control file modules.

User Action: If the maximum task size for the partition is less than 32K words, use the MCR command SET /MAXEXT or DCL command SET SYSTEM/EXTENSION_LIMIT to increase the maximum task size, or run RPT in a different partition. If this occurs while generating summaries for large numbers of packets, try reducing the amount of data needed by using RPT switches to reduce the number of packets analyzed for each summary.

ERLRPT-F-MEMINIFAI, Memory allocation initialization failure.

Explanation: RPT could not dynamically extend its address space to create room for its data structures.

User Action: If the maximum task size for the partition is less than 32K words, use the MCR command SET /MAXEXT or DCL command SET SYSTEM/EXTENSION_LIMIT to increase the maximum task size, or run RPT in a different partition.

ERLRPT-F-MODLOAGRP, Undefined group referenced by module to be loaded.

Explanation: The control file module being loaded attempted to reference an undefined group.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MODLOASYM, Undefined symbol in module to be loaded.

Explanation: The control file module being loaded attempted to reference an undefined symbol.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MODNAMENL, Module name cannot be null.

Explanation: A control file module attempted to access another control file module that had a null or blank name.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MODNOMEM, Insufficient free memory to load module.

Explanation: RPT could not dynamically extend its address space to create room for control file modules.

User Action: If the maximum task size for the partition is less than 32K words, use the MCR command SET /MAXEXT or DCL command SET SYSTEM/EXTENSION_LIMIT to increase the maximum task size, or run RPT in a different partition. If this occurs while generating summaries for large numbers of packets, try reducing the amount of data needed by using other switches to reduce the number of packets analyzed for each summary.

ERLRPT-F-MODSTART, Starting module for execution not found.

Explanation: The control file library must contain a module named DISPATCH.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-MODZERO, Attempt to modulus by zero.

Explanation: A control file module attempted to perform a MOD by zero.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-NOMORESTK, Execution stack overflow.

Explanation: RPT's execution stack has overflowed.

User Action: Edit RPTBLD.CMD to increase the extension for program section XCSTK0, and rebuild RPT.

ERLRPT-F-NOSTACKE, Internal error - Pop from execution stack with empty stack.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-NOTDYNFIL, Dynamic file operation performed on invalid group.

Explanation: A control file module specified a group that was not defined as DYNAMIC_TABLE in a statement or operation requiring a DYNAMIC_TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-NOTPOINT, POINTER LOAD or MOVE executed with a non-pointer variable.

Explanation: A control file module executed a POINTER LOAD or MOVE with a variable that was not a pointer.

User Action: Correct the user-written module or submit an SPR for, DIGITAL-supplied modules.

ERLRPT-F-NOTPOIVAR, POINTER LOAD with no pointer variable specified.

Explanation: A control file module executed a POINTER LOAD or MOVE with no variable specified.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-NUMINVOPR, Invalid numeric double-operand operation code.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-OPRINVLOG, Attempt to perform logical operation on an invalid type.

Explanation: A control file module attempted to perform a logical operation with operands that were neither NUMERIC nor LOGICAL.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-OPRNOTIMP, Operation not implemented.

Explanation: A control file module attempted to perform a multiplication where both operands were larger than a word value.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-PACKETSIZ, Illegal packet size.

Explanation: The size of an error log packet was zero or would cause the packet to cross a block boundary.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-POISETGRP, POINTER variable is not from correct group in POINTER ... LOAD or MOVE.

Explanation: A control file module executed a POINTER LOAD or MOVE statement in which the optional pointer variable was not a pointer to the specified DYNAMIC_TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-POISETMOD, POINTER variable is from wrong module in POINTER ... LOAD or MOVE.

Explanation: A control file module executed a POINTER LOAD or MOVE statement in which the DYNAMIC_TABLE pointed to by the optional pointer variable was not in the same module as the DYNAMIC_TABLE specified in the POINTER statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-POISETSIZ, Group too small for POINTER in POINTER ... LOAD or MOVE.

Explanation: A control file module executed a POINTER LOAD or MOVE statement in which the optional pointer variable was pointing past the end of the specified DYNAMIC_TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-PROCNAMEN, Null procedure name.

Explanation: A control file module specified a null or blank procedure name in a CALL or ENABLE statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-RAD50BYTE, Cannot convert a byte using Radix-50 conversion.

Explanation: A control file module attempted to convert an ASCII string or numeric literal to a BYTE using Radix–50 conversion.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-RELINVCOD, Invalid relational operator.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-RETURNNOC, A RETURN was executed with no corresponding CALL.

Explanation: A control file module executed a RETURN statement outside of a procedure or coroutine.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-SELECTNOM, SELECT statement index has no matching statement block.

Explanation: A control file module executed a SELECT statement with no statement block to match the value of the numeric control expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-SIGNOTASC, Parameter or message in SIGNAL-class statement not ASCII.

Explanation: A control file module executed a SIGNAL, SIGNAL_STOP, or MESSAGE statement with a non-ASCII argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-SIGTOOBIG, Message and parameters in SIGNAL-class statement too long.

Explanation: A control file module executed a SIGNAL, SIGNAL_STOP, or MESSAGE statement in which the length of the concatenated message and parameters was longer than 255 characters.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-SIGTOOMAN, Cannot issue a SIGNAL during SIGNAL processing.

Explanation: A control file module executed a SIGNAL or SIGNAL_STOP statement while processing a previous SIGNAL or SIGNAL_STOP.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-STANOTIMP, Statement not implemented.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-STANOTVAL, Internal error - invalid statement code.

Explanation: This is an internal error within RPT.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-SUBEXTBIG, Substring extraction end element exceeds string.

Explanation: A control file module attempted to perform a substring extraction in which the substring exceeded the end of the string.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-SUBPKTSIZ, Illegal subpacket size.

Explanation: The current subpacket exceeded the bounds of the packet.

User Action: Please submit an SPR with any information you have.

ERLRPT-F-UNDEFPROC, Specified procedure not found.

Explanation: A control file module has executed a CALL statement, and the specified procedure was not found.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-UNDMODULE, Specified module not found.

Explanation: A control file module has executed a CALL statement, and the specified module was not found.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-VALSTKOVR, Value stack overflow.

Explanation: The stack used for processing values and expressions has overflowed.

User Action: Edit RPTBLD.CMD to increase the extension for program section VLSTK0, and rebuild RPT.

ERLRPT-F-VALUESIZE, Value in expression is too large.

Explanation: A control file module evaluated an expression in which an intermediate value or the final value was too large.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-VALUETYPE, Value in expression is wrong type.

Explanation: A control file module evaluated an expression in which an intermediate value or the final value was of the wrong type.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-VARNOCONT, Attempt to access variable in group without context.

Explanation: A control file module attempted to reference a variable for a TABLE, DYNAMIC_TABLE, or PACKET or SUBPACKET with the REPEATED attribute for which the current record context was not valid.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-VARNODATA, Attempt to access variable in group with no data.

Explanation: A control file module attempted to reference a variable for a TABLE, DYNAMIC_TABLE, or PACKET or SUBPACKET with no data.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-VARNOTDAT, Attempt to load data into a BIT or FIELD variable.

Explanation: A control file module attempted to load a value into a BIT or FIELD in a group, rather than into the data item for which the BIT or FIELD was defined.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLRPT-F-WRITEACCV, Attempt to load a value into a non-writeable variable.

Explanation: A control file module attempted to load a value into a data item in a PACKET, SUBPACKET, or TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

Chapter 4
Error Log Control File Architecture

This chapter describes the architecture of error log control files. A knowledgeable system programmer can use the information presented here to add user-written modules to the error logging system.

This chapter includes the following sections:

- Section 4.1 defines the most important terms and concepts presented in this chapter.
- Section 4.2 describes the control file modules, the flow of program control through the modules, module compilation paths, and how to recompile modules after making modifications.
- Section 4.3 describes the interaction between a dispatcher module and a device-level module.
- Section 4.4 explains event-level and device- or CPU-level dispatching.
- Section 4.5 provides the information you need to include error logging support for non-DIGITAL devices.
- Section 4.6 includes annotated listings of source code for three modules: ERM23, DSP2P1, and NRM23. The source code is keyed to discussions in Sections 4.3 and 4.5 of this chapter.

4.1 Terms and Concepts

The following are definitions of the most important terms and concepts presented in this chapter:

Control file	A collection of modules that together perform a function, such as processing error log files.
Module	A component of the error logging system. There are three kinds of modules: source modules, which have the file type CNF, object modules, which have the file type ICF, and listing modules, which have the file type LST.
Control File Language	The language in which control files are written. The Control File Language (CFL) is described in Chapter 5.
Error log file	The file that contains the raw error logging data. One record in the file corresponds to one event. The default specification for this file is LB:[1,6]LOG.ERR.
Event	Something that is logged in the error log file. An event may be the recording of an actual device error or it could be some informational data, such as a device mount or a change in system time.
Packets, subpackets	Each record (or event) is also a packet. A packet begins with a length word and is followed by data, which can consist of zero or more subpackets. A subpacket also consists of a length word followed by data. Every packet in the error logging system contains at least one subpacket.

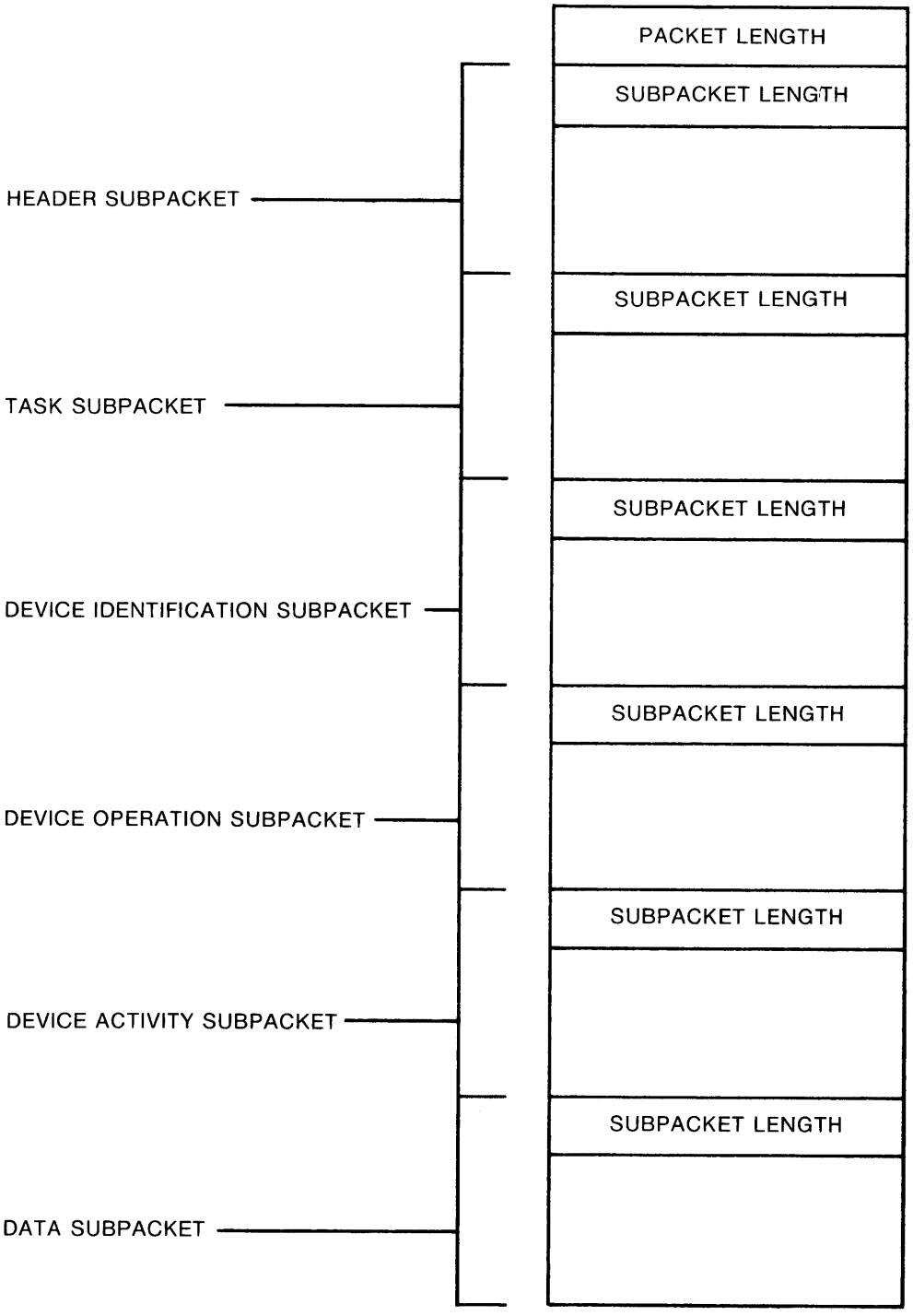
Note

The packet length word begins the packet, but it is not part of the packet; the packet length word is kept by FCS. Therefore, the packet length word is not included in the length of the packet. However, the subpacket length words are part of the packet and are included in its length. This is consistent throughout the error logging system. See Figure 4–1 for the general structure of an error logging packet.

4.2 Control File Module Architecture

The error logging system is modular; that is, information and dependencies specific to different devices are isolated in modules written for each device. This section describes the architecture of the control file modules: the modules themselves, the flow of program control through the modules, the compilation paths, and how you can modify and recompile the modules.

Figure 4–1: Structure of Error Logging Packet



ZK-1111-82

4.2.1 The Control File Modules

The following descriptions briefly explain the function of each of the control file modules:

DISPATCH

The DISPATCH module (the root module for the error logging system) declares all commonly used variables, calls the INITM1 module to initialize the system, and calls the PARSEM module to obtain and parse the command line. DISPATCH then requests the records from the input log file, declares the common subpackets (HEADER, TASK, DEVICE_ID, DEVICE_OPERATION, and DEVICE_ACTIVITY) for each record, computes the correct dispatcher module name, and calls that module. When all the records are processed, it calls the summary modules (if requested) and then calls the FINLP1 module to clean up.

See Section 4.6.4 for the definitions of the standard DIGITAL subpackets. Dispatching is described in more detail in Section 4.4.

PARSEM

The PARSEM module declares variables local to the processing of the command line and calls the PARS1M module to obtain the command line. It then calls the PARS2M module to process any switches and the PARS3M module to open the various files. PARSEM also provides commonly used parsing routines to the other parsing modules.

PARS1M

The PARS1M module initializes parsing variables and gets the command line from RPT. It then breaks all of the file specifications out of the command line, leaving all of the switches. PARS1M then searches for the /REPORT switch. If it finds the switch, PARS1M replaces it with the specified string of predefined switches.

PARS2M

The PARS2M module gets a switch from the string of switches produced by the PARS1M module. It then checks the switch for ambiguity and calls the PRS2AM module to process the switch. If PRS2AM does not recognize the switch, it is passed to the PRS2BM module. PARS2M repeats this process until all switches have been processed.

PRS2AM

The PRS2AM module processes the following switches: /DATE, /DEVICE, and /PACKET.

PRS2BM

The PRS2BM module processes the following switches: /FORMAT, /SERIAL, /SUMMARY, /TYPE, /VOLUME, and /WIDTH.

PARS3M

The PARS3M module applies the default values to any switches that were not specified and opens the specified files.

SELTM1

The SELTM1 module is called by the DISPATCH module to determine if the current packet meets the selection criteria of the command line switches.

DSP1P1

The DSP1P1 module processes error log control events (see Section 4.4.1). These modules declare the DATA subpacket for each type of event and process the event to completion, calling the formatter modules to print the common data if the FULL report format is specified.

DSP2P1

The DSP2P1 module processes device error events (see Section 4.4.1). These modules call the DEVSM1 module to determine the name of the device-level module required to process the event and then calls that module as a coroutine and passes control to it. The device-level module declares the DATA subpacket and then extracts information from the registers of the logged device so it can provide additional selection information.

When the device-level module returns control to DSP2P1, it performs the last of the selection tests and makes the decision whether to continue with this event or not. If DSP2P1 decides to continue, and if the FULL report format has been specified, DSP2P1 calls the formatter modules to print the common information. Once printing is completed, control returns to the device-level module, which prints the device registers.

If the BRIEF report has been specified, DSP2P1 still must decide whether to continue, but there is no need for the formatter modules: DSP2P1 does its own printing.

Once all the printing is completed, the error logging system tests to see if this entry belongs in any summaries. If so, DSP2P1 places the data relevant to each requested summary in the summary files.

DSP3P1

The DSP3P1 module processes device information events (see Section 4.4.1). It performs the same function as the DSP2P1 module, but only for device errors not related to I/O. This module is required only if you have a TU78 or Mass Storage Control Protocol (MSCP) device.

DSP4P1

The DSP4P1 module processes device control information events (see Section 4.4.1). DSP4P1 calls the DEVSM1 module to get the type of device associated with the device mnemonic.

Mount, dismount, and reset operations have no DATA subpacket. The formatter modules print the information if the FULL report mode is specified; otherwise, the module does all the printing itself. As with the DSP2P1 and DSP3P1 modules, DSP4P1 records summary information if requested.

The block replacement event does have a DATA subpacket, which is processed entirely by this module. This type of event does not contribute to summaries.

DSP5P1

The DSP5P1 module processes events detected by the CPU (see Section 4.4.3). DSP5P1 gets the CPU type from the HEADER subpacket declared by DISPATCH and calls the appropriate CPU-level module as a coroutine if the event was a memory parity error. The processing then proceeds much like that for device errors.

If the event was an unknown interrupt, the module declares and processes the DATA subpacket itself.

DSP6P1

The DSP6P1 module processes system control information events (see Section 4.4). There is no DATA subpacket associated with the power recovery event. DSP6P1 calls the formatter modules to print the common data if the FULL report format is specified; otherwise, the module does all the printing itself.

DSP7P1

The DSP7P1 module processes control information events (see Section 4.4). This module declares the DATA subpacket for each type of event and processes the event to completion, calling the formatter modules to print the common data if the FULL report format is specified.

DSP8P1

The DSP8P1 module processes control information events (see Section 4.4). This module uses the DATA subpacket to extract information for unknown interrupts. It also formats the output for unknown interrupt CPU-detected error packets.

FINLP1

The FINLP1 module is called by the DISPATCH module to perform cleanup operations after all the error log events are processed.

FINLP1 also creates the final page of the error log report. This page contains such information as the command line entered by the user, the files used, the switch states, the number of events processed, and how long it took to generate the report.

FMTNP1 and FMTWP1

FMTNP1 and FMTWP1 are formatter modules that print the first page of a full report; that is, all of the information from the HEADER, TASK, DEVICE_ID, DEVICE_OPERATION, and DEVICE_ACTIVITY subpackets. FMTNP1 prints reports in narrow format, and FMTWP1 prints reports in wide format.

DEVSM1

The DEVSM1 module is called by the DSP2P1, DSP3P1, and DSP4P1 modules to provide certain device-related information. DSP2P1 and DSP3P1 call it to find, among other things, the name of the device-level module that should help process the event. DSP4P1 calls DEVSM1 to find out the name of the device associated with a device mnemonic.

If the device mnemonic is DU, DEVSM1 then calls DEVUDA to do most of the processing.

DEVUDA

The DEVUDA module is called only by DEVSM1. It assists DEVSM1 in the processing of events on MSCP DU-type devices.

ERRORM

The ERRORM module is the error processor for the error logging system. Whenever a SIGNAL or SIGNAL_STOP procedure occurs, ERRORM processes the error.

SMRYEP

The SMRYEP module prints ERROR summaries. DISPATCH calls SMRYEP after all packets have been processed if an ERROR summary was requested.

SMRYGP

The SMRYGP module prints GEOMETRY summaries. DISPATCH calls SMRYGP after all packets have been processed if a GEOMETRY summary was requested.

SMRYHP

The SMRYHP module prints HISTORY summaries. DISPATCH calls SMRYHP after all packets have been processed if a HISTORY summary was requested.

CPU-level modules

There are four CPU-level modules, all with names derived from their associated processors. They are called as coroutines by DSP5P1 to process memory parity errors. The modules are as follows:

E1144	Processes errors from the PDP–11/44 processor
E117X	Processes errors from the PDP–11/70 and PDP–11/74 processors
E118X	Processes errors from the PDP–11/73, PDP–11/83, and PDP–11/84 processors
E11XX	Processes errors from all other PDP–11 processors

EUNKWN

EUNKWN is a universal device-level or CPU-level module. EUNKWN is called if a particular device-level module is unavailable or if the device mnemonic is unknown to the error logging system. EUNKWN is also called if the CPU type is unknown.

EUNKWN produces a formatted dump of the data, showing the relative offset within the data, and the data itself in octal word, octal high-byte, octal low-byte, and binary word format. See Section 4.5 for more information on error logging from unknown devices.

DMPALL

The DMPALL module is similar to EUNKWN. DMPALL is called if the packet cannot be processed due to an error in format or structure. DISPATCH calls DMPALL if the packet fails any sanity check.

DMPALL produces a formatted dump of the data, showing the relative offset within the data, and the data itself in octal word, octal high-byte, octal low-byte, and binary word format.

Device-level modules

Device-level modules contain details of the bit-to-text translation for all supported error logging devices. The DSP2P1 and DSP3P1 modules call them as coroutines. Their names are derived from the names of the associated devices.

See Section 4.5 for information on how these modules are constructed and how you can write device-level modules for unsupported devices.

Table 4–1 lists the standard error logging device-level modules.

Table 4–1: Error Logging Device-Level Modules

Module Name	Module Description
EML11	Processes ML11 errors
ERK05	Processes RK03 and RK05 errors
ERK67	Processes RK06 and RK07 errors
ERL12	Processes RL01 and RL02 errors
RM05	Processes RM05 errors
ERM23	Processes RM02 and RM03 errors
ERM80	Processes RM80 errors
ERP07	Processes RP07 errors
ERP23	Processes RP02 and RP03 errors
ERP456	Processes RP04, RP05, and RP06 errors
ERS11	Processes RS11 errors
ERS34	Processes RS03 and RS04 errors
ERX01	Processes RX01 errors
RX02	Processes RX02 errors
ET0310	Processes TS03, TE10, and TU10 errors
ET1645	Processes TE16, TU16, and TU45 errors
ETA11	Processes TA11 errors
ETC11	Processes TC11 errors
ETK50	Processes TK50 errors
ETK70	Processes TK70 errors
ETS11	Processes TS11, TU80 errors
ETSV05	Processes TSV05 errors
ETU58	Processes TU58 errors
ETU77	Processes TU77 errors
ETU81	Processes TU81 errors
MSCP5X	Processes errors for RD50, RD51, RD52, RD53, RD54, RX33, and RX50 devices
MSCP60	Processes MSCP RA60 errors
MSCP80	Processes MSCP RA80, RA81, and RA82 errors
MSCPAT	Processes MSCP attention errors

Table 4–1 (Cont.): Error Logging Device-Level Modules

Module Name	Module Description
MSCPCE	Processes MSCP controller errors
MSCPEN	Processes MSCP end packets
MSCPSD	Processes MSCP RC25 errors
MSCPTO	Processes MSCP timeout errors
TMSCPE	Processes TMSCP end packets

Notes modules

Notes modules contain notes for error conditions that need additional explanation. Notes modules are device-specific and have names derived from the names of the associated device-level module. See Section 4.5.3.2 for more information on how these modules are constructed.

Table 4–2 lists the standard error logging notes modules.

Table 4–2: Error Logging Notes Modules

Module Name	Module Description
NML11	Processes ML11 notes
NRK67	Processes RK06 and RK07 notes
NRM05	Processes RM05 notes
NRM23	Processes RM02 and RM03 notes
NT0310	Processes TS03, TE10, and TU10 notes
NTS11	Processes TS11 and TU80 notes

4.2.2 Program Control Flow

The general flow of program control through the control file modules is as follows:

1. RPT opens the control file. In most cases, this is the default control file LX or LB:[1,6]ERRLOG.ULB. If you wish to use some other filespec, RPT must be rebuilt to prompt for the new name of the file.
2. RPT creates the module table in its dynamic work space. This table contains an entry for each module in the control file universal library.
3. RPT loads the DISPATCH module and transfers control to the ENTRY procedure.
4. The ENTRY procedure is similar to the root module in most MACRO–11 programs. The general flow of the ENTRY procedure is as follows:
   a. Declares most of the commonly used data structures.
   b. Enables the ERROR_1 procedure in the ERRORM module as an error handler.

c. Fetches and parses the command line by calling the SETUP procedure in the PARSEM module.

d. Performs some general initialization with the INIT_1 procedure from INITM1.

e. Sets up a loop to step through the PACKET_RANGE file, extracting pairs of packet ranges which are fed back to RPT.

f. Loops through the current packet range, requesting each packet in turn and calling the DISPATCH procedure in the DISPATCH module. This step is performed for each packet range.

g. When all packets and packet ranges have been requested, ENTRY generates summaries, if requested, by calling the SUM_ERROR, SUM_GEOMETRY, and SUM_HISTORY procedures from SMRYEP, SMRYGP, and SMRYHP, respectively.

h. Calls the FINAL_1 procedure in FINLP1 to perform cleanup operations.

5. The DISPATCH procedure in the DISPATCH module declares all of the common subpackets. In order of appearance, these are the HEADER, TASK, DEVICE_ID, DEVICE_OP, and IO_ ACTIVITY subpackets.

   Each of these subpackets has a mask bit in the HEADER subpacket to indicate the presence of the subpacket. If the bit is set, the subpacket is present and therefore declared. If the bit is not set, the subpacket is not present and consequently not declared.

   Note that the HEADER subpacket must always be present.

   As each subpacket is declared, various tests are performed that must be passed. (These tests are for the various selection criteria that you can specify using command line qualifiers.) If one subpacket fails the tests, the entire packet is rejected. If the tests are passed, the procedure then computes the name of the appropriate dispatcher module.

   The dispatcher module name is concatenated from the following elements:

   - The string "DSP"
   - The event code (HEADER.CODE_TYPE) converted to ASCII decimal
   - The string "P1"

   For example, an event with a code of 5 would be dispatched to the DSP5P1 module.

6. The dispatcher modules (or the modules they may call) handle the declaration of the DATA subpacket if one is present. The dispatcher modules also perform further selection tests as needed to determine whether or not information about the event should be printed. Brief format reports are printed entirely by the dispatcher module. Full and register format reports are printed by a combination of the following modules:

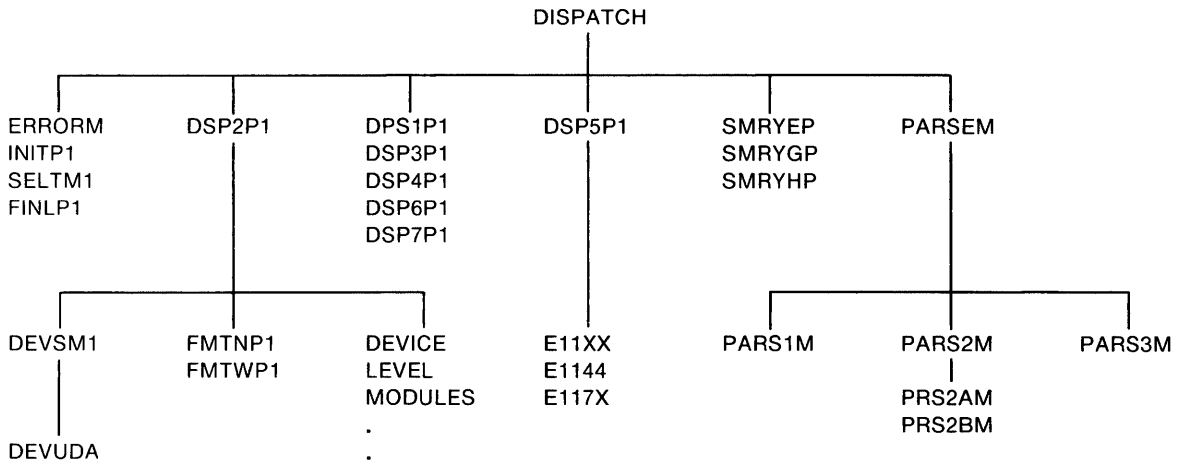
   - One of the formatter modules (FMTNP1 or FMTWP1)
   - The appropriate dispatcher module
   - A device-level module (if it is a device error) or a CPU-level module (if it is a processor or memory error)

4.2.3 Compilation Paths

The DISPATCH module must be compiled first. The next modules to be compiled are at the next level. These modules — ERRORM, DSP2P1, DSP1P1, DSP5P1, SMRYEP, and PARSEM — all use the symbol file produced from the compilation of DISPATCH as input. Modules in the same group (for example, ERRORM, INITP1, SELTM1, and FINLP1) all use the same input symbol file (in this case, DISPATCH) and can be compiled in any order.

Figure 4–2 illustrates the compilation paths for the control file modules.

Figure 4–2: Compilation Path for Control File Modules



ZK-1113-82

Where modules in the figure are connected by vertical lines, the upper module is compiled first. The lower module or modules are then compiled using the symbol file produced by the module at the next higher level. Therefore, the DSP2P1 module is compiled using the symbol file from DISPATCH, the DEVSM1 module is compiled using the symbol file from DSP2P1, and so on.

Modules used in the error logging system must be compiled using the following compile-time literal declarations:

```
Option>LITERAL SUPPORT.RSX_11M = FALSE
Option>LITERAL SUPPORT.RSX_11M_PLUS = TRUE
Option>LITERAL SUPPORT.IO_ACTIVITY = TRUE
Option>/
```

See Chapter 5 for a description of the Control File Language (CFL), which is used in these declarations.

4.2.4 Modification and Recompilation

You can modify any control file module. After doing so, you must recompile the module and replace it in the control file library.

Caution

You must recompile all modules on the same branch of the tree at levels lower than the modified module if your modification includes any of the following changes:

- Creates new groups, tables, or dynamic tables
- Creates a new variable within any of these structures
- Reorders a variable within any of these structures

Note that you need not recompile lower-level modules if you modify the run-time logic. For example, suppose you want to modify the following statement in the PARSEM module:

```
IF %STR$LENGTH(PARSE.SWITCH_LIST) EQ 0
```

Your modification is as follows:

```
IF %STR$LENGTH(PARSE.SWITCH_LIST) EQ 1
```

This modification in the PARSEM module would not require recompiling any of the lower-level modules (PARS1M, PARS2M, PARS3M, PRS2AM, or PRS2BM).

Now, suppose you modify the same statement as follows:

```
IF %STR$LENGTH(PARSE.SWT_LIST) EQ 1
```

In this example, you created the variable SWT_LIST in the PARSE group. Therefore, you must recompile the lower-level modules. This is because the information in the symbol file consists of group names (in alphabetical order) and the variables defined within the group (in the order declared). The compiler uses the information from the input symbol file to compute relative group and variable numbers for use when a module references a group or variable declared in a higher-level module. These group and variable numbers (rather than the names) are used to resolve references to groups and variables when a module is loaded. Defining new groups, or variables within a group, changes the relative order of these symbols.

4.3 Interaction Between Dispatcher and Device-Level Modules

The following section describes in detail the interaction between a dispatcher module and a device-level module, using the processing of an RM03 error as an example.

The following discussion refers to the ERM23 device-level module code in Section 4.6.1 and the DSP2P1 dispatcher module code in Section 4.6.2. Both the discussion here and the code in those two sections are keyed to each other by the module names (either ERM23 or DSP2M1) and numbers that look like this: ❶.

You may wish to remove the pages for Sections 4.6.1 and 4.6.2 from your book for easier reference in following the interaction between these two modules.

Processing in DSP2P1 begins when the DISPATCH module has declared all subpackets except for the DATA subpacket. All subpackets except for the TASK subpacket are needed for a device error.

DSP2P1 first declares the logical variable INDICATE.TAPE_FLAG (DSP2P1 ❶). This variable is set by DEVSM1 to indicate whether or not the device is a magnetic tape. The variable is used later in processing summary information.

DSP2P1 determines that peripheral errors are requested and that the subcode is valid. Having completed these checks, DSP2P1 calls DEVSM1 (DSP2P1 ❷). DEVSM1 returns device information in the following three variables:

- INTERMOD_DEVERR.DISP_NAME contains the name of the device-level module needed to process the DATA subpacket, in this case ERM23.
- INTERMOD_DEVERR.DRIVE_TYPE contains the string RM03 for the drive type.
- INTERMOD_DEVERR.ALT_NAME contains the string RM02/03 for the alternate drive type. (The alternate name variable is not used during device error processing.)

After returning from DEVSM1, the NOTE_NUMBERS file is cleared (DSP2P1 ❸). Clearing the file deletes any records that may remain there from previous events.

The next step establishes the coroutine relationship with the device-level module (DSP2P1 ❹). The DEVICE_ERROR procedure in the DSP2P1 module is one partner, while the DEVICE_ENTRY procedure in the ERM23 module is the other. Control passes to the DEVICE_ERROR procedure.

The first thing the DEVICE_ERROR procedure does is pass control to its partner (DSP2P1 ❺). Module ERM23 receives control at the beginning of the DEVICE_ENTRY procedure (ERM23 ❷).

The DEVICE_ENTRY procedure declares the DATA subpacket (ERM23 ❸). Once this is completed, the INTERMOD_DEVERR variables are filled in (ERM23 ❻ and ❼). All of the variables must be filled in. If the information for a particular variable is unavailable or not applicable, use the string N/A. For the variable INTERMOD_DEVERR.ERROR_CYLINDER, the string ??? also has a special meaning: It indicates to DSP2P1 that the section titled Device Error Position Information is to be suppressed.

Once the INTERMOD_DEVERR variables are all filled in, a coroutine statement returns control to DEVICE_ERROR (ERM23 ❽). DEVICE_ERROR regains control where it left off (DSP2P1 ❻).

DSP2P1 then performs the serial number tests, if required, after having first initialized the variable INTERMOD_DEVERR.REJECT_FLAG to false. If the serial number test is failed, the variable INTERMOD_DEVERR.REJECT_FLAG is set to true.

The path through the two modules now depends on the report format, either BRIEF, FULL, REGISTER, or NONE. The following paragraphs explain these paths:

BRIEF

The REJECT_FLAG variable is tested (DSP2P1 ❼). If true, there is no output. If false, one line is printed which contains the information required for a BRIEF report. In either case, the variable INTERMOD_DEVERR.PRINT_FLAG is set to false.

If the packet is not rejected and if the report format is not NONE, the variable REPORT. PRINT_COUNT is incremented. This variable keeps a count of how many events were printed (as opposed to how many were looked at, which is a separate tally).

FULL

The REJECT_FLAG variable is tested (DSP2P1 ❽). If it is set to true, there is no output and the PRINT_FLAG variable is set to false.

If REJECT_FLAG is set to false, the formatter modules are called to print the information from the common subpackets. The DSP2P1 module then prints (still on the first page) the information passed back in the INTERMOD_DEVERR variables filled in by ERM23. DSP2P1 generates a page break, then prints a header on the second page. When done, DSP2P1 sets the PRINT_FLAG variable to true.

REGISTER

The REGISTER path is almost identical to the FULL path. The only difference is that the page containing all of the common information is not printed. The header on the page containing the register translation supplies a summary of the information instead.

NONE

The NONE path sets the PRINT_FLAG variable to false (DSP2P1 ❾).

All of these paths converge again at DSP2P1 (❿), where control once again passes to the DEVICE_ENTRY procedure in ERM23.

The first thing the DEVICE_ENTRY procedure does upon regaining control is to test the PRINT_FLAG variable (ERM23 ❾). If it is set to FALSE, the module exits (ERM23 ⓫).

If the PRINT_FLAG variable is set to true, ERM23 performs the bit-to-text translation of the registers. Following that, any required notes are indicated by PUT statements to the NOTE_NUMBERS file specifying the note index (ERM23 ❿). The module then exits (ERM23 ⓫).

When ERM23 exits, the DSP2P1 module regains control and the coroutine partnership is broken (DSP2P1 ⓫).

After the device-level module has printed (if instructed to) and has exited back to the DSP2P1 module, the UPDATE_RECORD procedure in DSP2P1 is called (DSP2P1 ⓬).

The UPDATE_RECORD procedure tests to see if an ERROR summary was requested (DSP2P1 ⓭). If none was requested, processing goes on to the GEOMETRY section.

If an ERROR summary was requested, the DSP2P1 module searches the ERROR_INFO_E file to see if an error having the same error type has been encountered. If the same error type has been encountered, the record in the file describing that type of error is updated to show that one more error occurred, and when it occurred. If no such error is found in the file, a new record that describes the error is added to the file and processing goes on to the GEOMETRY section.

The UPDATE_RECORD procedure then tests to see if a GEOMETRY summary was requested (DSP2P1 ⓮). If not, the procedure exits.

Updating the ERROR_INFO_G file is much the same as updating the ERROR_INFO_E file, but the information recorded is somewhat different. In particular, the GEOMETRY summary indicates where on the device the error occurred. Since magnetic tapes have no valid geometry information, when updating the ERROR_INFO_G file you need to know whether or not the device is a magnetic tape.

Finally, the DEVICE_ERROR procedure checks for entries in the NOTE_NUMBERS file. If there are any entries, DSP2P1 computes the name of the notes file. The name of a notes module is the same as its corresponding device-level module, except that the first character of the module name is the letter N. In this case, the module name is NRM23. The notes module is called to print the requested notes.

4.4 Dispatching

This section discusses module dispatching. There are two levels of dispatching: event-level dispatching and device- or CPU-level dispatching.

4.4.1 Event-Level Dispatching

All events that occur in the error logging system are assigned a unique combination of code and subcode. These code and subcode combinations can be found in the file EPKDF.MAC (EPKDF$ macro in EXEMC.MLB) along with the definition of the structure of error log packets. See Appendix C for a listing of EPKDF$. Table 4–3 summarizes the error logging code and subcode combinations.

Table 4–3: Error Logging Code and Subcode Combinations

Code	Subcode
1. Error Log Control	1. Error Log Status Change
	2. Switch Logging Files
	3. Append File
	4. Declare Backup File
	5. Show (not logged)
	6. Change Limits
2. Device Errors	1. Device Hard Error
	2. Device Soft Error
	3. Device Interrupt Timeout (hard)
	4. Spurious Interrupt
	5. Device Interrupt Timeout (soft)
3. Device Information	1. Device Information Message
4. Device Control Information	1. Device Mount
	2. Device Dismount

Table 4–3 (Cont.): Error Logging Code and Subcode Combinations

Code	Subcode
	3. Device Counts Reset
	4. Block Replacement
5. Memory_Info Errors	1. Memory Error
6. System Control Information	1. Power Recovery
7. Control Information	1. Time Change
	2. System Crash
	3. Device Driver Load
	4. Device Driver Unload
	5. Message
8. CPU-Detected Errors	2. Unexpected Interrupt

Each code group is processed by one of the dispatcher modules. These modules are named DSP1P1 through DSP7P1. The name of the dispatcher module is concatenated in the DISPATCH module's DISPATCH procedure from the following elements:

- The string "DSP"
- The event code (HEADER.CODE_TYPE) converted to ASCII decimal
- The string "P1"

The single-digit ASCII conversion of the code value (obtained from the HEADER subpacket) is required because the Librarian Program Utility (LBR) allows a maximum of six Radix–50 characters for a module name. The code value 9 is currently unused; values 0 and 8 are reserved.

Once the dispatcher module has been called, it checks to see if this type of event was requested. If the event type was not requested, the module returns, effectively ignoring the entry. Event types are requested by using the /TYPE switch. The event types, codes, and the dispatcher modules that process them are listed in Table 4–4.

Table 4–4: Event Types, Codes, and Their Dispatcher Modules

Type	Codes	Dispatcher Modules
ALL	1-7	DSP1P1, ..., DSP7P1
CONTROL	1	DSP1P1
ERRORS	2,3,5	DSP2P1, DSP3P1, DSP5P1
MEMORY	5	DSP5P1
PERIPHERAL	2,3	DSP2P1, DSP3P1
PROCESSOR	5	DSP5P1
SYSTEM_INFO	4,6,7	DSP4P1, DSP6P1, DSP7P1
CPU_INFO	8	DSP8P1

Once the dispatcher module determines that this type of event was requested, it checks to see if the subcode is in range. If it is not, the event is rejected with an error message.

At this point, dispatcher modules may declare and print the DATA subpacket themselves or they may call lower-level modules to do so. The error logging dispatcher modules handle all of the printing for the brief report mode. If the full report mode is specified, the dispatcher modules call one of the following modules to print the common portions of the event:

Width	Formatter Module
NARROW	FMTNP1
WIDE	FMTWP1

The dispatcher module may print the rest of the event itself or work with a lower-level module.

4.4.2 Device-Level Dispatching

Device-level dispatching is performed with the assistance of the DEVSM1 module. This module is called by the DSP2P1, DSP3P1, and DSP4P1 modules and determines, among other things, the correct device-level module for the event.

The following paragraphs describe how the DEVSM1 module works (see the source code for exceptional cases; this discussion addresses only common cases).

The first thing DEVSM1 checks is whether there is a DEVICE_ID subpacket. If no DEVICE_ID subpacket is found, an error results. Once past that check, DEVSM1 uses the device mnemonic to search the DEVICE_INFO table. If the device is not found, DEVSM1 specifies the EUNKWN module in the variable INTERMOD_DEVERR.DISP_NAME.

Assuming that the mnemonic is recognized, the DEVSM1 module tests to see if (a) the mnemonic is that of a MASSBUS device, and (b) there is a DATA subpacket. Assuming both are true, DEVSM1 looks ahead into the DATA subpacket to obtain the MASSBUS Drive Type from the logged registers.

The drive-type value is unique for each MASSBUS device. Once this value is obtained, the DEVICE_INFO table is searched again, this time using the drive-type value as the key. If this search succeeds, the INTERMOD_DEVERR.DISP_NAME variable is filled in with the module name specified by the resulting record in the table.

If there is no DATA subpacket, or if the device is not a MASSBUS device, the search of the table is left pointing to the first record that matched on the specified mnemonic. DEVSM1 performs a further search of the table based on the mnemonic as well as the device size (which is provided in the DEVICE_ID.DEV_TYPE variable). The INTERMOD_DEVERR.DISP_NAME variable is then filled in with the module name specified in the record (the result of this search).

The search proceeds in the following order:

1. When a device's mnemonic is found in the DEVICE_INFO table in module DEVSM1, the MASSBUS_FLAG is checked. If it is true, a look-ahead into the device registers returns the device's DRIVE_TYPE.
2. The DEVICE_INFO table is then searched again to find a record having that drive type.
3. The error logging system then dispatches to the module corresponding to the actual registers logged, rather than dispatching to the module indicated by the mnemonic provided in the Executive.

For MASSBUS devices, the error logging system uses the device name provided by the DEVICE_INFO table. This name will always be correct, since each MASSBUS device has a unique drive-type value. If there is a mismatch between the mnemonic supplied and the device type as determined by examining the registers, the device-type field in the printed report is preceded by an asterisk (*).

You make the error logging system aware of a new device-level module by adding a record to the DEVICE_INFO table in the DEVSM1 module. A section of the table is reproduced in Table 4–5.

Table 4–5: The DEVICE_INFO Table

```
TABLE DEVICE_INFO;
MNEMONIC        :ASCII [2] ;        ! Device mnemonic
PRINT_NAME      :ASCII [6] ;        ! Name for printing
ALT_PRINT_NAME  :ASCII [12] ;       ! Alternate name for printing
DISP_NAME       :ASCII [6] ;        ! Name of device module
SIZE            :LONGWORD ;         ! Size of device
MASSBUS_FLAG    :LOGICAL ;          ! True if a MASSBUS device
DRIVE_TYPE      :BYTE ;             ! MASSBUS device-type number
BEGIN_TABLE
```

Table 4–5 (Cont.): The DEVICE_INFO Table

'CT', 'TU60',	'TU60',	'ETA11',	#LD'0',	FALSE,	#BO'0' ;
'DB', 'RP04',	'RP04/05',	'ERP456',	#LD'171798',	TRUE,	#BO'20' ;
'DB', 'RP05',	'RP04/05',	'ERP456',	#LD'171798',	TRUE,	#BO'21' ;
'DB', 'RP06',	'RP06',	'ERP456',	#LD'340670',	TRUE,	#BO'22' ;
'DD', 'TU58',	'TU58',	'ETU58',	#LD'512',	FALSE,	#BO'0' ;
'DF', 'RF11',	'RF11',	'ERS11',	#LD'-1',	FALSE,	#BO'0' ;
'DK', 'RK05',	'RK03/05',	'ERK05',	#LD'4800',	FALSE,	#BO'0' ;
'DL', 'RL01',	'RL01',	'ERL12',	#LD'10240',	FALSE,	#BO'0' ;
'DL', 'RL02',	'RL02',	'ERL12',	#LD'20480',	FALSE,	#BO'0' ;
'DM', 'RK06',	'RK06',	'ERK67',	#LD'27126',	FALSE,	#BO'0' ;
'DM', 'RK07',	'RK07',	'ERK67',	#LD'53790',	FALSE,	#BO'0' ;
'DP', 'RP03',	'RP03',	'ERP23',	#LD'80000',	FALSE,	#BO'0' ;
'DR', 'RM02',	'RM02/03',	'ERM23',	#LD'131680',	TRUE,	#BO'25' ;
'DR', 'RM03',	'RM02/03',	'ERM23',	#LD'131680',	TRUE,	#BO'24' ;
'DR', 'RM05',	'RM05',	'RM05',	#LD'500384',	TRUE,	#BO'27' ;
'DR', 'RM80',	'RM80',	'ERM80',	#LD'242606',	TRUE,	#BO'26' ;
'DR', 'RP07',	'RP07',	'ERP07',	#LD'1008000',	TRUE,	#BO'42' ;
'DS', 'RS03',	'RS03/04',	'ERS34',	#LD'1024',	TRUE,	#BO'0' ;
'DS', 'RS03',	'RS03/04',	'ERS34',	#LD'1024',	TRUE,	#BO'1' ;
'DS', 'RS04',	'RS03/04',	'ERS34',	#LD'2048',	TRUE,	#BO'2' ;
'DS', 'RS04',	'RS03/04',	'ERS34',	#LD'2048',	TRUE,	#BO'3' ;

Table 4–5 (Cont.): The DEVICE_INFO Table

'DT', 'TU56',	'TU56',	'ETC11',	#LD'576',	FALSE,	#BO'0' ;
.					
.					
.					

The columns of the table, read from left to right, correspond to the declared items MNEMONIC, PRINT_NAME, ALT_PRINT_NAME, DISP_NAME, SIZE, MASSBUS_FLAG, and DRIVE_TYPE. The following descriptions explain each of these declared items:

MNEMONIC

The mnemonic is a 2-character ASCII field that is listed as the device mnemonic in the Device Control Block (DCB). Records should be kept in alphabetical order by mnemonic.

PRINT_NAME

This 6-character ASCII field identifies the particular device. This field is used in the printing of the Device Identification Information section of FULL or REGISTER reports whenever the device registers are available. Normally this field is used, unless devices are being mounted or dismounted. In such cases, the device registers are not available and, depending on the device, there may be insufficient information to completely identify a device. When this occurs, the ALT_PRINT_NAME field is used instead.

ALT_PRINT_NAME

This 12-character ASCII field identifies the device when the device registers are not available, usually for mounts and dismounts. When this happens (depending on the device), there may be insufficient information to identify a device completely. For example, when an RP04 is mounted, the only information available that can identify the device is the mnemonic DB and the device size. This information is the same for an RP04 and an RP05. In this case the ALT_PRINT_NAME field is used, which identifies the device as an RP04/05.

DISP_NAME

This 6-character ASCII field identifies the name of the device-level module used to process error logging entries for the particular device.

SIZE

This longword specifies the number of blocks on the device. There are two special values associated with this field, as follows:

- A value of zero (0) indicates that the device is a magnetic tape.
- A value of –1 indicates there is no fixed size for the device. DEVSM1 will not correctly handle combinations of fixed- and variable-size devices having the same mnemonic.

MASSBUS_FLAG

This logical value indicates whether or not the device is a MASSBUS device. It is set to true for MASSBUS devices and false for any other devices.

DRIVE_TYPE

This byte specifies the MASSBUS drive-type value. Each MASSBUS device has a unique value which is available in the low byte of the drive-type register. If the record is not for a MASSBUS device, this field should be zero (0).

4.4.3 CPU-Level Dispatching

CPU-level dispatching is performed by DSP5P1. The HEADER subpacket contains a variable called PROC_TYPE that indicates the type of processor the error was logged on. DSP5P1 uses that variable to search a table that contains module names associated with the CPU-type value.

4.5 Support of Non-DIGITAL Devices

This section explains what you have to do to provide error logging support for non-DIGITAL devices.

Adding error logging support for a non-DIGITAL device consists of either one or three of the following steps, depending on the desired level of support:

1. The first step is to include error logging support in the driver. Without this support, no information can be logged for the device. For full error logging support, you must perform the following additional steps.
2. Write a device-level module for the new device.
3. Add the new module to the control file library and make the error logging system aware of the new module.

4.5.1 Error Logging of Unknown Devices

The error logging system can handle entries from devices unknown to the system. Entries from an unknown device are handled by the EUNKWN module, which functions as a universal device-level module. For a brief report, EUNKWN will pass back the value N/A in the INTERMOD_DEVERR variables to indicate that the information is not available. For a full report, EUNKWN prints the device registers in a dump-style format where the bit-to-text translation would normally take place. The rest of the report is unchanged.

4.5.2 Providing Driver Support for a Non-DIGITAL Device

The Executive module ERROR contains the routines to be used by a driver to log device errors. A device error in this sense can be a real error, a timeout, or perhaps an informational message. The following sections discuss the routines in general. See the code in [11,10]ERROR.MAC for more detail.

4.5.2.1 The $BMSET Routine

The $BMSET coroutine raises the processor priority to 7 (to lock out interrupts), sets the S2.ACT interrupt active bit in S.ST2 of the SCB, and then calls the caller to start the I/O function. When the re-called caller returns, $BMSET lowers the processor priority to 0, thus allowing interrupts once again.

Parameters

Input

R4 = SCB address

Output

The S2.ACT interrupt active bit is set.

4.5.2.2 The $DVTMO and $DTOER Routines

The $DVTMO routine logs device timeouts at PR0, and the $DTOER routine logs device timeouts at device priority. The routines behave identically except that $DTOER disables the device interrupt and lowers the processor priority to 0.

The routines set S2.ACT, the interrupt active bit in the SCB word S.ST2. They then test to see if the timeout is a diagnostic function. Diagnostic functions are never logged.

The routines load the error code and subcode in R0 and the routine passes control to the $DVCER routine.

Parameters

Input

R2 = Address of a block of registers to log (must be the CSR address if KS.MBC is set)

R4 = SCB address

R5 = UCB address

Output

R0 = IE.DNR and 377 (Device not Ready)

R1 = I/O packet address

C = 0, if not a diagnostic function.
User response: Create an error log packet and fill it in. Put a pointer to the packet (S.BMSV) in the SCB and set the error in progress bit SP.EIP in S.PRI.

C = 1, if a diagnostic function.
User response: Do not create an error log packet.

4.5.2.3 The $DVERR ($DVCER) Routine

$DVERR and $DVCER are the same routine; $DVCER is the routine name, and $DVERR is a synonym. This routine logs device errors. If an error is already in progress on the device, it will be ignored. If not, $DVCER allocates an error log packet and fills it in with the context of the current transfer. Note that this routine requires that there be an I/O packet associated with this error. See Section 4.5.2.6 to log an error where there is no I/O active on the device.

Note that the system also logs information about concurrent I/O activity on other devices.

Parameters

Input

R2	=	Address of a block of registers to log (must be the CSR address if KS.MBC is set)
R4	=	SCB address
R5	=	UCB address

Output

If no error is already in progress on this device, allocate an error log packet, fill it in, point the SCB to the packet, and set the error in progress bit.

If an error is in progress on this device, this routine is a no-op.

4.5.2.4 The $NSIER Routine

The $NSIER routine logs nonsense interrupts.

When you assign interrupt vectors for a device, you do not expect interrupts from the remaining, unused vectors. The $NSIER routine allows you to detect interrupts from unused vectors and log the errors as nonsense interrupts.

Parameters

Input

@(SP)	=	Contains bits 06:04 of the unused vector number.

Output

If a nonsense interrupt is in the process of being logged, increment the interrupt count.

If this is the beginning of the processing of a nonsense interrupt, identify the vector and create and queue an error log packet.

4.5.2.5 The $FNERL Routine

The $FNERL routine is called at I/O completion or when it is necessary to queue an error log packet after a successful recovery of a mid-transfer error. This routine effectively completes the processing of an error.

The routine first inserts the error retry information. It then tests to see if this was a hard (unrecoverable) error or a soft (recoverable) error and updates the packet accordingly. (All errors are assumed to be hard up to this point.) Depending on the result of that test, $FNERL tests against the appropriate limit to see if the limit has already been met. If the limit had been previously met, the packet is discarded. If not, $FNERL updates the appropriate error count, logs the packet, and sets the SCB to show that the processing of this error has been completed.

Parameters

Input

R0	=	First I/O status word
R2	=	Starting and final error retry counts (if 0, do not update limits)
R3	=	Error log packet address (if R4 = 0)
R4	=	SCB address or 0
R5	=	UCB address

Output

Either queue or discard the error log packet (depending on the limits) and set the SCB to indicate that no error is being processed.

4.5.2.6 The $LOGER Routine

Drivers use the $LOGER routine to create an error log packet when no I/O is present, such as when a driver receives an unsolicited interrupt from a device that contains information that should be logged. $LOGER creates the packet normally, but the driver is responsible for filling in the DATA subpacket information. Otherwise, processing is similar to the $DVERR routine.

Parameters

Input

R1	=	Length of data to be logged (in bytes)
R4	=	SCB address (if 0, then no I/O packet is present)
R5	=	UCB address

Output

C	=	1; error cannot be logged for some reason
C	=	0; error can be logged
R1	=	Address of DATA area in the packet
R3	=	Address of error log packet

4.5.2.7 The LOGTST Routine

The LOGTST routine is not for use by drivers. Other routines in the ERROR module call LOGTST to see if an error can or should be logged.

4.5.2.8 The $CRPKT Routine

The $CRPKT routine creates an error log packet. It is called as part of the $SMSG directive processing, and by other Executive routines as part of the processing of a memory error, nonsense interrupt, time change, powerfail recovery, or device error.

The $CRPKT routine determines the required format and size of the packet, allocates the required amount of pool, and then fills in the packet. It obtains information from SYSCOM, the appropriate DCBs, UCBs, SCBs, TCBs, VCBs, and the I/O packet, as required.

Note that a HEADER subpacket is always required. A forced system failure will result if $CRPKT detects the "no HEADER subpacket" condition.

If a driver of an error logging device calls the $CRPKT routine to create an error logging packet, the data address for the data subpacket must not be an address within the driver. Specifically, the address must not be mapped by APR 5, as this APR is used to map the common. Any user-written driver that performs such a function must allocate a piece of pool, fill in the appropriate information, and pass the pool address to the $CRPKT routine.

Note that information about concurrent I/O activity on other devices is always logged in addition to the activity on the device in question.

Parameters

Input

R0	=	Packet code and subcode (see EPKDF for details)
R1	=	Length of DATA subpacket
R2	=	Control mask word (see EPKDF for details)
R3	=	Beginning address of data for DATA subpacket
R4	=	TCB address (for TASK subpacket)
R5	=	UCB address (for DEVICE_ID subpacket)

Output

R0	=	Unchanged
R1	=	Beginning address of data in the DATA subpacket
R2	=	Unchanged
R3	=	Beginning address of error log packet
R4	=	Unchanged
R5	=	Unchanged
C	=	0 if a packet was created
C	=	1 if a packet was not created

4.5.2.9 The CALDEV Routine

The CALDEV routine calculates the logical unit number for the given UCB.

Parameters

Input

R0	=	Pointer into the error log packet
R3	=	DCB address
R5	=	UCB address

Output

(R0)	=	Unit number stored in the error log packet at the byte (R0)
R0	=	Updated to point to next byte in error log packet

4.5.2.10 The $QUPKT Routine

The $QUPKT routine queues an error log packet. If there is no other packet in the queue, $QUPKT requests the Error Logger task (ERRLOG) with a delay of 2 seconds. If there is another entry in the queue, $QUPKT requests ERRLOG to run immediately. Command packets from ELI always cause ERRLOG to run immediately.

Parameters

Input

R3	=	Pointer to packet for insertion in queue

Output

None

4.5.2.11 The $QERMV Routine

The $QERMV routine removes an entry from the error log queue and transfers it to a user buffer. It is called only by ERRLOG.

Parameters

Input

R4	=	Length of user buffer
R5	=	Address of user buffer

Output

R1	=	Length of packet (if R1 $\neq$ 0)
R4	=	Unchanged
R5	=	Unchanged
C	=	0 if packet was successfully removed
C	=	1 if there was no packet to remove or packet was too long. If R1 $\neq$ 0, the packet was too long and R1 contains the packet length. If R1 = 0, then there was no packet to remove.

4.5.3 Providing Error Logging Support for a Non-DIGITAL Device

In addition to requiring that you correctly provide driver support, full error logging support for a non-DIGITAL device requires the following two extra steps:

1. Write the device-level module for the new device. This module contains the detailed instructions on how to interpret the logged information; that is, the bit-to-text translation information for the device registers. The information common to all events is interpreted by the DIGITAL-supplied modules.
2. Add the new module to the control file library and make the error logging system aware of the new module.

4.5.3.1 Writing a Device-Level Module

This section explains the general structure of device-level modules, using the RM02/03 module ERM23 as an example. Section 4.6.1 is an annotated listing of ERM23 and Section 4.6.3 is an annotated listing of the notes module for the RM02/03 driver. Both the discussion here and the code in those two sections are keyed to each other by the module names (either ERM23 or DSP2P1) and numbers that look like this: ❶.

You may wish to remove the pages for Sections 4.6.1 and 4.6.3 from your book for easier reference in following the interaction between these two modules.

In general, the flow of a device-level module proceeds as follows:

❶ MODULE statement followed by module header

❷ PROCEDURE statement

❸ SUBPACKET declaration

❹ Register definitions

❺ Declaration of local work variables and table declarations

❻ Intermodule variable loading

❼ Error-type determination

❽ Coroutine back to caller

❾ Bit-to-text translation and register printing

❿ Note requirements indicated

⓫ Exit the module

Each of these procedures is described in the following subsections.

The MODULE Statement ❶

The name of a user-written module must be in the following form:

```
MODULE ExxUSR ident
```

Generally, the module name begins with the letter E, followed by five or fewer letters indicating the device or devices served by the module. For example, the ERM23 module handles the RM02 and RM03 disks, while the ERP456 module serves the RP04, RP05, and RP06 disks.

The letters xx are the 2-character device mnemonic. Your device mnemonic cannot be the same as any DIGITAL-supplied device mnemonic.

The ident field is an identification value that is stored in the module. Usually, the ident begins with a letter that identifies the operating system the module is intended to be used with (such as P for RSX–11M–PLUS), followed by a version and update number in the standard DIGITAL style.

The module header follows. This includes the copyright statement, author, date written, and audit trails of modifications.

The PROCEDURE Statement ❷

The PROCEDURE statement for a user-written module must be in the following form:

```
PROCEDURE DEVICE_ENTRY
```

The procedure name must be DEVICE_ENTRY. This name is hard-coded into the DSP2P1 and DSP3P1 dispatcher modules.

The SUBPACKET Declaration ❸

The device-level module is responsible for the declaration of the device data (usually in the form of registers). The SUBPACKET declaration defines the number of registers, how they are printed, and the bit-to-text translations for the various bits and fields of the registers.

The following code example is a segment of the ERM23 device-level module for the RM02/03 disk drives:

```
SUBPACKET subpacket_name = DISP.NEXT_PACKET NAMED ;

    reg_name:       WORD MACHINE ;
       :            BIT [15]:       'true_text' ;
       :            BIT [14]:       'true_text',
                                    'false_text' ;

    aux_label:      FIELD [12:2]:   'Bits 12 and 13 = '
                                    | %CNV_$BINARY(Subpacket_name.aux_label, 2, '0')
                                    | ' (B)' ;
       :            BIT [11]:       'true_text' ;
                      .
                      .
                      .
    reg_name:       WORD MACHINE ;
                      .
                      .
                      .
END_PACKET ;
```

The following comments explain some of the key elements in the preceding example:

- The subpacket name is usually REGISTER, although this name is not required.
- DISP.NEXT_PACKET is a variable that contains the subpacket number of the data subpacket and has been set up by the preceding modules.
- The NAMED qualifier indicates to RPT that the register labels are to be saved for later printing.
- Following the SUBPACKET declaration are the definitions of the registers and their bits and fields.
- The statement END_PACKET indicates the end of the subpacket declaration.

Register Definitions ❹

The label assigned to a register provides both a reference to the register (a variable name) and a name for the register when printing. The register name is printed later on (if you specified a full format report). In most cases, the error logging system uses the same register names used by DIGITAL Field Service hardware documents.

For the RM02/03 device-level module, the first register declared looks like the following:

```
RMCS1:      WORD MACHINE ;
  :         BIT [15]:       '*Special Condition set' ;

RMCS1_TRE:  BIT [14]:       '*Transfer Error' ;
  :         BIT [13]:       '*MASSBUS Control Bus Parity Err' ;
  :         BIT [12]:       '*Unused bit set' ;
  :         BIT [11]:       ' Drive Available',
                            '*Drive not Available (other port using it)' ;
  :         BIT [10]:       ' UNIBUS B Selected for Data Transfer',
                            ' UNIBUS A Selected for Data Transfer' ;

RMCS1_BA:   FIELD [8:2]:    ' BA17,BA16 = '
                            | %CNV_$BINARY(REGISTER.RMCS1_BA, 2, '0')
                            | ' (B)' ;
  :         BIT [ 7]:       ' Controller Ready',
                            ' Controller not Ready' ;
  :         BIT [ 6]:       ' Interrupt Enabled',
                            ' Interrupt not Enabled' ;
              .
              .
              .
```

The following comments explain some of the key elements in the preceding example:

- The first line indicates that the name of the register is RMCS1 and that it is a word (16 bits) in length.
- The MACHINE qualifier states that, when printed, the register is to be formatted in the native radix for the machine that the report is being generated on. The native radix for a PDP–11 processor is octal, and for a VAX processor, hexadecimal. Other print qualifiers are available to change the radix, such as HEX, OCTAL, DECIMAL, BCD, BINARY, and RAD50.
- The second line defines bit 15 of the register RMCS1, including when it is to be printed and what is to be printed. Only one text string is provided. This indicates that the bit is to be printed only when set to true. Otherwise, nothing is printed for that bit.
- Bit 14 has the label RMCS1_TRE. Labels assigned to bits and fields are never printed. They are allowed so you can reference the bit or field as a variable. As with bit 15, the text for this bit is printed only if the bit is set to true.
- Bit 11 has two text arguments. The first argument is printed if the bit is set and the second argument is printed if the bit is reset. In other words, this bit will always be printed.
- Bits 8 and 9 are defined to be a FIELD with the variable name RMCS1_BA. The general format for a field definition is as follows:

```
FIELD [starting_bit_number:number_of_bits]: 'other_string',
                                            '0_string',
                                            '1_string',
                                            '2_string',
                                                 .
                                                 .
                                                 .
                                            'N_string' ;
```

- In the field definition, the 0_string is printed if the value of the field is 0. The 1_string is printed if the value of the field is 1, and so on. The other_string is printed if the field has a value that has no corresponding text string. Note that for the field RMCS1_BA there is only an other_string. Therefore, this field is always printed.

  A technique that is used in the DIGITAL device-level modules is to declare a field over any contiguous unused bits. The other_string is defined to be 'Unused bits set', and the 0_string is defined to be null (the null, or zero_length, string). If the field has the value 0, nothing is printed. If, however, any of the bits are set, the field appears in the report.

  Note that all of the text strings associated with bits and fields have as their first character either a space or an asterisk. When printing the text for a bit or field, RPT removes the first character of the string and places it in front of the bit or field position indicator. An asterisk signals some kind of special condition. For example, bit 11 of RMCS1 can print one of the following ways:

  ```
  [11] Drive Available

  *[11] Drive not Available (other port using it)
  ```

  Remember that the asterisk does not necessarily indicate an error, just something interesting. A blank in front of the position indicator means a normal or status condition.

- You can use IF...THEN...ELSE, CASE, and SELECT statements to conditionalize the declaration of the subpacket. The statement blocks in these structures must be enclosed by BEGIN and END. You can use variables previously declared in the subpacket even though the declaration of the subpacket is not complete. Also note the use of the %LOK (look-ahead) functions in various device-level modules. They look into a subpacket before it is declared, usually to produce variables to control the declaration.

- Note the variable REGISTER.LENGTH towards the end of the subpacket declaration in ERM23. This variable was created when the SUBPACKET statement was executed. The variable name is of the form subpacket_name.LENGTH and contains the number of bytes in the subpacket.

Declaration of Local Work Variables and Tables ❺

The device-level module often needs some local variables and tables. These are usually defined after the end of the subpacket declaration, although this is not required. However, remember that variables must be declared in a module before they can be used.

If the device specified in your module is a disk, you must set the INDICATE.TAPE_FLAG to false. If the device is not a disk, set the INDICATE.TAPE_FLAG to true.

Loading of the Intermodule Variables ❻

The DISPATCH module declares a collection of variables having the group name INTERMOD_DEVERR. Some of these ASCII string variables pass information from the device-level modules back to their caller. The variables that must be filled in are as follows:

- INTERMOD_DEVERR.DRIVE_SN
- INTERMOD_DEVERR.DEV_FUNCTION
- INTERMOD_DEVERR.PHYS_UNIT
- INTERMOD_DEVERR.ERROR_CYLINDER
- INTERMOD_DEVERR.ERROR_SECTOR
- INTERMOD_DEVERR.ERROR_HEAD
- INTERMOD_DEVERR.ERROR_GROUP
- INTERMOD_DEVERR.BLOCK_NUMBER
- INTERMOD_DEVERR.ERROR_TYPE
- INTERMOD_DEVERR.DRIVE_TYPE (see Section 4.5.3.3 for more details on this variable)

This section of the module is where these variables are filled in. Use the string N/A if the information is either not applicable or not available. Note that for certain devices, most notably magnetic tapes, the ERROR_CYLINDER variable is filled in with the ??? string. This flag tells the dispatcher module to suppress the printing of the section entitled Device Error Position Information. Note that one of the variables to be filled in contains the error type. See the following subsection for more details on how the error type is determined.

Determination of the Error Type ❼

The error-type definition is a determination of the most likely problem as indicated by the error bits for a given event. It is not a determination of what failed, but rather an indication of what events occurred that could be associated with the failure. The error type is determined solely on the basis of the bits in the current event; no interevent analysis is performed.

The error type is determined by a precedence parse of the various error bits found in the device registers. The DECODE statement, in conjunction with IF...THEN...ELSE-type constructs, is used to search the bits in a specific order. The first condition found to be true stops the search.

Coroutine Back to Caller ❽

Once all the intermodule variables have been filled in, a coroutine statement returns control to the device module's caller. The caller examines the returned information and determines whether to continue processing the event. Nothing has been printed up to this point in the processing of this event.

If the decision is not to proceed, but to reject the event, the caller (a) sets the INTERMOD_DEVERR.PRINT_FLAG variable to false, and (b) returns control, through a coroutine statement, to the device-level module.

If the decision is to proceed, the caller performs some or all of the printing, depending on whether the print format is FULL or BRIEF. If the FULL format is specified, the caller (a) prints everything except the device registers, (b) sets the INTERMOD_DEVERR.PRINT_FLAG variable to true, and (c) returns control, through a coroutine statement, to the device-level module. If the format is brief, the caller (a) performs all required printing, (b) sets the INTERMOD_DEVERR.PRINT_FLAG variable to false, and (c) returns control, through a coroutine statement, to the device-level module.

When the device-level module regains control, it examines the print flag. If the flag is set to true, the module prints the device registers and generates any required note indicators. If the print flag is false, the module exits.

Perform the Bit-to-Text Translation and Register Printing ❾

If the INTERMOD_DEVERR.PRINT_FLAG variable is true, the device-level module prints the device registers and performs the required bit-to-text translation. This is done by executing a WRITE statement (to produce column headers) followed by a WRITE_GROUP statement. The WRITE_GROUP statement references the subpacket name specified in the SUBPACKET statement. It also uses two variables, REPORT.W_G_F_1 and REPORT.W_G_F_2, as format strings. These variables are initialized by the INITM1 module and contain the format strings for printing the register data in either wide or narrow format. If you need to print data that does not conform to the formats defined by these variables, you can define your own format. You can test the logical variable REPORT.WIDE to determine whether a wide or narrow report was requested.

If the INTERMOD_DEVERR.PRINT_FLAG variable is false, the device-level module exits.

Indicate Any Notes That Are Required ❿

The error logging system can print notes for certain conditions that need additional explanation. If you need such notes, you can create a notes module (see Section 4.5.3.2 for details) and include it in the library. You can then request a note by referencing it from the device-level module.

You request a note by inserting a PUT statement in the NOTE_NUMBERS file that specifies the note number in the NOTE_NUMBERS.INDEX variable. For example, the RM02/03 device-level module can optionally generate a note if certain unused bits in the RMDA register are set. This is done by using the following code:

```
!
! If the unused bits  5 to 7 are set in the RMDA register.
!
IF (REGISTER.RMDA [5:3] NE #BB'0')
THEN
    !
    ! Print the note saying that it may cause an invalid
    ! sector address to be recognized resulting in a
    ! possible invalid address error.
    !
    PUT NOTE_NUMBERS INDEX = 1 ;
END_IF ;
```

When the device-level module exits, the caller tests to see whether any notes were requested. If notes were requested, the dispatcher strips the first character from the device-level module's name and replaces it with the letter N. For example, the notes module for ERM23 (the RM02/03 device-level module) is NRM23. The dispatcher calls the notes module, which determines which notes were requested and prints them.

Multiple notes can be requested. They are printed in the order requested.

Exit the Module ⓫

When everything is done, the device-level module exits. Exiting a module implies a RETURN instruction to the module's caller. Exiting from a device-level module also breaks the coroutine relationship.

4.5.3.2 Writing a Notes Module

This section explains the structure of a notes module using the RM02/03 notes module as an example. Section 4.6.3 contains an annotated listing of this module.

The general flow of a notes module is as follows:

❶ MODULE statement followed by module header

❷ PROCEDURE statement

❸ Notes heading

❹ Selection of a note for printing

❺ Handling of an unknown note number

❻ Getting the next note

❼ Exit the module

The MODULE Statement ❶

The MODULE statement for a notes module must be in the following form:

```
MODULE NxxUSR ident
```

The module name of a notes module is related to its corresponding device-level module name by replacing the first letter of the device-level module's name with the letter N to get the notes module name. This convention must be followed because the notes module name is derived from the name of the device-level module and is never looked up in a table.

See Section 4.5.3.1 for an explanation of the ident field of the MODULE statement.

The PROCEDURE Statement ❷

The PROCEDURE statement for a notes module must be in the following form:

```
PROCEDURE NOTES
```

The procedure name must be NOTES. This is coded into the DSP2P1 and DSP3P1 dispatcher modules.

The Notes Heading ❸

The notes heading declares what is about to be printed. Notice that notes appear directly following the register interpretation in FULL and REGISTER reports only.

Selecting a Note for Printing ❹

Notes are selected for printing by testing the NOTE_NUMBERS file for context after performing a POINTER NOTE_NUMBERS FIRST operation. If records remain (that is, if there is context), a SELECT statement is performed on the variable NOTE_NUMBERS.INDEX. This variable indicates which note to print.

Handling an Unknown Note Number ❺

The ELSE clause of the SELECT statement traps unknown note numbers. A SIGNAL operation is performed using the UNKNWNNOT error indication. The note number and the drive type are passed to the error handler as string arguments.

Getting the Next Note ❻

The next note is obtained by POINTER NOTE_NUMBERS NEXT. This causes RPT to point to the next record in the NOTE_NUMBERS file. If another record exists, the NOTE_NUMBERS file has context at the top of the WHILE...DO loop. If no other record exists, there will be no context; hence, there will be no more notes.

Exit the Module ❼

When everything is done, the notes module exits. Exiting a module implies a RETURN instruction to the module's caller.

4.5.3.3 MASSBUS and Non-MASSBUS Device Considerations

All device-level modules work essentially the same way; the only exception is that MASSBUS modules are not required to fill in the INTERMOD_DEVERR.DRIVE_TYPE variable, whereas non-MASSBUS modules must fill in this variable.

This requirement has to do with mixed MASSBUS configurations. With mixed configurations, the Executive's database may not match the actual configurations if unit plugs have been inadvertently swapped.

To avoid unwanted results in case of such a mismatch, the error logging system performs the following operations:

1. When a device's mnemonic is found in the DEVICE_INFO table in module DEVSM1, the MASSBUS_FLAG is checked. If it is true, a look-ahead into the device registers returns the device's DRIVE_TYPE.
2. The DEVICE_INFO table is then searched again to find a record having that drive type.
3. The error logging system then dispatches to the module corresponding to the actual registers logged, rather than dispatching to the module indicated by the mnemonic provided in the Executive.

For MASSBUS devices, the error logging system uses the device name provided by the DEVICE_INFO table. This name will always be correct, since each MASSBUS device has a unique drive-type value. If there is a mismatch between the mnemonic supplied and the device type as determined by examining the registers, the device-type field in the printed report is preceded by an asterisk.

It is the device-level module's responsibility to supply correct drive-type information for non-MASSBUS devices. The DEVSM1 module fills in the value based on the device's mnemonic and size, but sometimes this information is not accurate. The RK03 and RK05 are examples of where this is necessary. Both RK03 and RK05 device errors are processed by the ERK05 module. The ERK05 module identifies the drive type by its device registers, and fills in the DRIVE_TYPE variable accordingly. Another example of where the correct drive-type information is necessary is DU devices, where the error logging system is only concerned that the device mnemonic be DU. It is up to the modules that handle these devices to provide the drive-type information.

4.6 Code Examples

This section provides examples of source code from the error logging system. These examples are annotated for use with previous sections of this chapter. They are written in the Control File Language (CFL), which is documented in Chapter 5. The examples in this chapter are as follows:

- The ERM23 device-level module for RM02 and RM03 drivers
- The DSP2P1 dispatcher module
- The NRM23 notes module for RM02 and RM03 drivers

4.6.1 The RM02/03 Device-Level Module ERM23

The following example is an annotated listing of ERM23.MAC, the device-level module for the RM02 and RM03 disk drives:

❶

```
MODULE ERM23 'M01.01' ;
!
! ERROR LOG CONTROL FILE MODULE: RM02, RM03
!
!                    COPYRIGHT (c) 1981 BY
!            DIGITAL EQUIPMENT CORPORATION, MAYNARD
!             MASSACHUSETTS.  ALL RIGHTS RESERVED.
!
! THIS  SOFTWARE  IS  FURNISHED  UNDER  A LICENSE AND MAY BE USED
! AND  COPIED  ONLY IN  ACCORDANCE WITH THE TERMS OF SUCH LICENSE
! AND WITH  THE INCLUSION  OF THE ABOVE  COPYRIGHT  NOTICE.  THIS
! SOFTWARE,  OR ANY OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
! OTHERWISE MADE  AVAILABLE TO ANY OTHER PERSON.  NO TITLE TO AND
! OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
```

```
! THE INFORMATION  IN THIS DOCUMENT IS SUBJECT  TO CHANGE WITHOUT
! NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT  BY  DIGITAL
! EQUIPMENT CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
! ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
!
! VERSION 01.01
!
! ROBERT E. LI                          08-JAN-81

! This is one of the many device modules, which is called by the device
! error dispatcher (DSP2P1) or device information dispatcher (DSP3P1)
! to process all the device dependent information.
!
! Modified by:
!
!            CBP                         Correct BAE/CS3 register logic
```

❷

```
PROCEDURE DEVICE_ENTRY
!
! This procedure,  which is called via a COROUTINE statement from a dispatch
! module,  declares and translates all  device registers or data fields of
! the data subpacket.  The intermodule variables required by the  dispatch
! modules are stuffed with the appropriate values, followed by a COROUTINE
! back to the dispatch module. The dispatch module then COROUTINEs back to
! this routine a second time,  at a point where a  write group  is used to
! print the details of a FULL or REGISTER report.

!
BEGIN
!
! Declare a variable to hold the length of the subpacket.
!
DECLARE PACKET_LENGTH ;
    TEMP            :BYTE ;
END_DECLARE ;
!
! Now get the length of the DATA subpacket. Remember that the returned value
! is in bytes and includes two bytes for the length word.
!
SET PACKET_LENGTH.TEMP TO %LOK_$LENGTH(DISP.NEXT_PACKET) ;
!
! Define the data subpacket offsets and all the print information.
!
```

❸

```
SUBPACKET REGISTER = DISP.NEXT_PACKET NAMED ;
```

❹

```
RMCS1:      WORD MACHINE ;
   :        BIT [15]:        '*Special Condition set' ;
RMCS1_TRE:  BIT [14]:        '*Transfer Error' ;
   :        BIT [13]:        '*MASSBUS Control Bus Parity Err' ;
   :        BIT [12]:        '*Unused bit set' ;
   :        BIT [11]:        ' Drive Available',
                             '*Drive not Available (other port using it)' ;
   :        BIT [10]:        ' Unibus B Selected for Data Transfer',
                             ' Unibus A Selected for Data Transfer' ;

RMCS1_BA:   FIELD [8:2]:     ' BA17,BA16 = '
                             | %CNV_$BINARY(REGISTER.RMCS1_BA, 2, '0')
                             | ' (B)' ;
   :        BIT [ 7]:        ' Controller Ready',
                             ' Controller not Ready' ;
   :        BIT [ 6]:        ' Interrupt Enabled',
                             ' Interrupt not Enabled' ;

RMCS1_FN:   FIELD [1:5]:     ' Function = '
                             | INTERMOD_DEVERR.DEV_FUNCTION ;
   :        BIT [ 0]:        '*Go bit on' ;
RMWC:       WORD MACHINE ;

   :        FIELD [0:16]:    %CNV_$DECIMAL_P(%COM_$NEGATE(REGISTER.RMWC), 6)
                             | ' words remaining' ;

RMBA:       WORD MACHINE ;
   :        FIELD [0:16]:    ' Bus Address Register' ;

RMDA:       WORD MACHINE ;
   :        FIELD [13:3]:    '*Unused bits set', NULL ;
RMDA_HD:    FIELD [8:5]:     ' Track Address  = '
                             | %CNV_$DECIMAL_P(REGISTER.RMDA_HD, 2) ;
   :        FIELD [5:3]:     '*Unused bits set (see note)', NULL ;

RMDA_SEC:   FIELD [0:5]:     ' Sector Address = '
                             | %CNV_$DECIMAL_P(REGISTER.RMDA_SEC, 2) ;

RMCS2:      WORD MACHINE ;
    :       BIT [15]:        '*Data Late' ;
RMCS2_WC:   BIT [14]:        '*Write Check Error' ;
    :       BIT [13]:        '*Parity Error';
    :       BIT [12]:        '*Nonexistent Drive' ;
    :       BIT [11]:        '*Nonexistent Memory' ;

    :       BIT [10]:        '*Program Error' ;
    :       BIT [ 9]:        '*Missed Transfer' ;
    :       BIT [ 8]:        '*MASSBUS Data Bus Parity Error' ;
    :       BIT [ 7]:        ' Output Ready (silo contains data)',
                             ' Output not Ready (silo empty)' ;
    :       BIT [ 6]:        ' Input Ready (silo not full)',

                             ' Input not Ready (silo full)' ;
    :       BIT [ 5]:        ' Controller Clear '
                             | '(clears all drives as well)' ;
    :       BIT [ 4]:        '*Parity Test set (even parity)',
                             ' Parity Test reset (odd parity)' ;
    :       BIT [ 3]:        '*Bus Address Increment Inhibit' ;
RMCS2_UN:   FIELD [0:3]:     ' Drive Selected = '
                             | INTERMOD_DEVERR.PHYS_UNIT ;
```

```
RMDS:       WORD MACHINE ;
   :        BIT [15]:       ' Attention Active' ;
RMDS_ERR:   BIT [14]:       '*Error (RMER1,2 have bits set)' ;
   :        BIT [13]:       ' Position in Progress' ;
   :        BIT [12]:       ' Medium Online', '*Medium not Online' ;
   :        BIT [11]:       ' Drive is Write Locked',
                            ' Drive is Write Enabled' ;
   :        BIT [10]:       ' Last Sector Transferred (last of the pack)' ;
   :        BIT [ 9]:       ' Programmable (ports program selectable)' ;
   :        BIT [ 8]:       ' Drive Present', '*Drive not Present' ;
   :        BIT [ 7]:       ' Drive Ready',
                            ' Drive not Ready' ;
   :        BIT [ 6]:       ' Volume Valid', '*Volume not Valid' ;
   :        FIELD [1:5]:    '*Unused bits set', NULL ;
   :        BIT [ 0]:       ' Drive in Offset Mode',
                            ' Drive not in Offset Mode' ;

RMER1:      WORD MACHINE ;
RMER1_DCK:  BIT [15]:       '*Data Check' ;
    :       BIT [14]:       '*Drive Unsafe' ;
    :       BIT [13]:       '*Operation Incomplete' ;
    :       BIT [12]:       '*Drive Timing Error' ;
    :       BIT [11]:       '*Write Lock Error' ;
    :       BIT [10]:       '*Invalid Address Error' ;
    :       BIT [ 9]:       '*Address Overflow Error' ;
    :       BIT [ 8]:       '*Header CRC Error' ;
    :       BIT [ 7]:       '*Header Compare Error' ;
RMER1_ECH:  BIT [ 6]:       '*ECC Hard Error' ;
    :       BIT [ 5]:       '*Write Clock Fail' ;
    :       BIT [ 4]:       '*Format Error' ;
    :       BIT [ 3]:       '*Parity Error' ;
    :       BIT [ 2]:       '*Register Modification Refused' ;
    :       BIT [ 1]:       '*Illegal Register' ;
    :       BIT [ 0]:       '*Illegal Function' ;

RMAS:       WORD MACHINE ;
   :        FIELD [8:8]:    %CND_$IF(REGISTER.RMDT [11],
                                    NULL,'*Unused bits set'),
                                    NULL ;
   :        BIT [ 7]:       ' Unit #7 Attention' ;
   :        BIT [ 6]:       ' Unit #6 Attention' ;
   :        BIT [ 5]:       ' Unit #5 Attention' ;
   :        BIT [ 4]:       ' Unit #4 Attention' ;
   :        BIT [ 3]:       ' Unit #3 Attention' ;
   :        BIT [ 2]:       ' Unit #2 Attention' ;
   :        BIT [ 1]:       ' Unit #1 Attention' ;
   :        BIT [ 0]:       ' Unit #0 Attention' ;

RMLA:       WORD MACHINE ;
   :        FIELD [11:5]:   '*Unused bits set', NULL ;
RMLA_ANG:   FIELD [6:5]:    ' Sector Count = '
                            | %CNV_$DECIMAL_P(REGISTER.RMLA_ANG, 2) ;
   :        FIELD [0:6]:    '*Unused bits set', NULL ;
```

```
RMDB:       WORD MACHINE ;
   :        FIELD [0:16]:    ' Data Buffer contents' ;

RMMR1:      WORD MACHINE ;
   :        BIT [15]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Debug Clock set', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Debug Clock reset', NULL) ;
   :        BIT [14]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Debug Clock Enabled', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Debug Clock Disabled', NULL) ;
   :        BIT [13]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Diagnostic End of Block set', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Diagnostic End of Block reset',NULL) ;
   :        BIT [12]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Search Time Out disabled', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Search Time Out enabled', NULL) ;
   :        BIT [11]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Clock set', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Clock reset', NULL) ;
   :        BIT [10]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Read Data set', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Read Data reset',NULL) ;
   :        BIT [ 9]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Unit Ready', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Unit Not Ready', NULL) ;
   :        BIT [ 8]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance On Cylinder', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance not On Cylinder',NULL) ;
   :        BIT [ 7]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 '*Maintenance Seek Error', NULL) ;
   :        BIT [ 6]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 '*Maintenance Drive Fault',NULL) ;
   :        BIT [ 5]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Sector Pulse set', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Sector Pulse reset', NULL) ;
   :        BIT [ 4]:        '*Unused bit set' ;
   :        BIT [ 3]:        %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Write Protect', NULL),
                             %CND_$IF(REGISTER.RMMR1_MM,
                                 ' Maintenance Write Enabled', NULL) ;
```

```
   :        BIT [ 2]:       %CND_$IF(REGISTER.RMMR1_MM,
                                ' Maintenance Index Pulse set', NULL),
                            %CND_$IF(REGISTER.RMMR1_MM,
                                ' Maintenance Index Pulse reset', NULL) ;

   :        BIT [ 1]:       %CND_$IF(REGISTER.RMMR1_MM,
                                ' Maintenance Sector Compare set', NULL),

                            %CND_$IF(REGISTER.RMMR1_MM,
                                ' Maintenance Sector Compare reset', NULL);
RMMR1_MM:   BIT [ 0]:       ' Diagnostic Mode on',
                            ' Diagnostic Mode off' ;

RMDT:       WORD MACHINE ;
   :        BIT [15]:       '*Drive not Sector Addressable' ;
   :        BIT [14]:       '*Unit is a Tape Drive' ;
   :        BIT [13]:       NULL, '*Unit is not a Moving Head Device' ;

   :        BIT [12]:       '*Unused bit set' ;
   :        BIT [11]:       ' DRQ on (dual port unit)',
                            ' DRQ off (single port unit)' ;
   :        FIELD [9:2]:    '*Unused bits set', NULL ;

RMDT_TYP:   FIELD [0:8]:    ' Drive Type = '
                            | INTERMOD_DEVERR.DRIVE_TYPE ;

RMSN:       WORD MACHINE ;
   :        FIELD [0:16]:   ' Drive Serial Number = '
                            | %CNV_$BCD(REGISTER.RMSN,4) | ' (BCD)' ;

RMOF:       WORD MACHINE ;
   :        FIELD [13:3]:   '*Unused bits set', NULL ;
   :        BIT [12]:       ' 16 Bit Data Format',
                            '*18 Bit Data Format' ;

   :        BIT [11]:       ' ECC Inhibit', ' ECC enabled' ;
   :        BIT [10]:       ' Header Compare Inhibit',
                            ' Header Compare Enabled' ;

   :        FIELD [8:2]:    '*Unused bits set', NULL ;
   :        BIT [ 7]:       ' Offset Direction = Forward',
                            ' Offset Direction = Reverse' ;

   :        FIELD [0:7]:    '*Unused bits set', NULL ;

RMDC:       WORD MACHINE ;
   :        FIELD [10:6]:   '*Unused bits set', NULL ;
RMDC_DC:    FIELD [0:10]:   ' Desired Cylinder = '
                            | %CNV_$DECIMAL_P(REGISTER.RMDC_DC, 4) ;

RMHR:       WORD MACHINE ;
   :        FIELD [0:16]:   ' Holding Register contents' ;

RMMR2:      WORD MACHINE ;
   :        BIT [15]:       ' Port A Request for Service' ;
   :        BIT [14]:       ' Port B Request for Service' ;
   :        BIT [13]:       ' Control Select Tag on' ;
   :        BIT [12]:       %CND_$IF(REGISTER.RMMR1_MM,
                            ' Test Sequencer Branching on', NULL) ;
```

```
        :       BIT [11]:       ' Control or Cylinder Tag on' ;
        :       BIT [10]:       ' Control or Head Tag on' ;

  RMMR2_MBL:    FIELD [0:10]:   %CND_$IF(REGISTER.RMMR1_MM,
                                ' Maintenance Bus Lines = '
                                | %CNV_$BINARY(REGISTER.RMMR2_MBL, 10, '0')
                                | ' (B)'
                                , NULL) ;

    RMER2:      WORD MACHINE ;
        :       BIT [15]:       '*Bad Sector Detected (hdr bit)' ;
        :       BIT [14]:       '*Seek Incomplete' ;
        :       BIT [13]:       '*Operator Plug Error (removed)' ;
        :       BIT [12]:       '*Invalid Command (VV bit reset)' ;

        :       BIT [11]:       '*Loss of System Clock' ;
        :       BIT [10]:       '*Loss of Bit Clock' ;
        :       FIELD [8:2]:    '*Unused bits set', NULL ;
        :       BIT [ 7]:       '*Device Check' ;

        :       FIELD [4:3]:    '*Unused bits set', NULL ;
        :       BIT [ 3]:       '*Data Parity Error' ;
        :       FIELD [0:3]:    '*Unused bits set', NULL ;

    RMEC1:      WORD MACHINE ;
        :       FIELD [13:3]:   '*Unused bits set', NULL ;

    RMEC1_PS:   FIELD [0:13]:   ' ECC Position = ' | VAR.ECCPS ;

    RMEC2:      WORD MACHINE ;
        :       FIELD [11:5]:   '*Unused bits set', NULL ;
        :       FIELD [0:11]:   ' ECC Pattern  = ' | VAR.ECCPAT ;

IF DEVICE_OP.FLG_BAE AND (PACKET_LENGTH.TEMP EQ #BD'46')
!
! If the RH70 flag is true and the packet length is 22 registers,
! declare the BAE and CS3 registers. Note that the packet length check
! is necessary because  unmapped  RSX systems will not log  BAE and CS3
! even if the controller is an RH70.

!
    THEN
    BEGIN
    RMBAE:      WORD MACHINE ;
        :       FIELD [6:10]:   '*Unused bits set', NULL ;
    RMBAE_EXT:  FIELD [0:6]:    ' BA21 through BA16 = '
                                | %CNV_$BINARY(REGISTER.RMBAE_EXT, 6, '0') ;

    RMCS3:      WORD MACHINE ;
        :       BIT [15]:       '*Address Parity Error' ;
        :       BIT [14]:       '*Data Parity Error, Odd Word' ;

        :       BIT [13]:       '*Data Parity Error, Even Word' ;
        :       BIT [12]:       '*Write Check Error, Odd Word' ;
        :       BIT [11]:       '*Write Check Error, Even Word' ;
        :       BIT [10]:       ' Double Word Transferred' ;

        :       FIELD [7:3]:    '*Unused bits set', NULL ;
        :       BIT [ 6]:       ' Interrupt Enabled',
                                ' Interrupt not Enabled' ;
        :       FIELD [4:2]:    '*Unused bits set', NULL ;
```

```
    RMCS3_IPC :FIELD [0:4]:     ' Inverse Parity Check Bits = '
                                | %CNV_$BINARY(REGISTER.RMCS3_IPC, 4, '0')
                                | ' (B)' ;
    END ;
    END_IF ;
END_PACKET ;
```

❺

```
!
! Declare all variables needed for the subpacket print information.
!
DECLARE VAR ;
    ECCPS:      ASCII [22] ;    ! ECC position.
    ECCPAT:     ASCII [22] ;    ! ECC pattern.
END_DECLARE ;

!
! Create the device function code conversion table.
!
TABLE FUNCTION ;
    FUN_CODE:       BYTE MACHINE ;
    FUN_TEXT:       ASCII [27] ;
BEGIN_TABLE
    #BO'00',       'No Operation' ;

    #BO'02',       'Seek Command' ;
    #BO'03',       'Recalibrate' ;
    #BO'04',       'Drive Clear' ;
    #BO'05',       'Release (dual port)' ;
    #BO'06',       'Offset Command' ;

    #BO'07',       'Return to Centerline' ;
    #BO'10',       'Read_in Preset' ;
    #BO'11',       'Pack Acknowledge' ;
    #BO'14',       'Search Command' ;
    #BO'24',       'Write Check Data' ;

    #BO'25',       'Write Check Header and Data' ;
    #BO'30',       'Write Data' ;
    #BO'31',       'Write Header and Data' ;
    #BO'34',       'Read Data' ;
    #BO'35',       'Read Header and Data' ;
END_TABLE ;

!
! Calculate the ECC Position.
!
! Determine if the  ECC position  is normal (not used), has an illegal
! value, points to the starting bit within the sector or is irrelevant.

!
IF REGISTER.RMEC1_PS LE #WD'4128'
THEN
    !
    ! At this point, the ECC position is within range  (0. to 4128.).
    ! Next, find out if the ECC position counter (register) was used.

    ! If the  ECC position register value  equals an octal 4066,  it
    ! indicates the register was initialized but not used.
    !
    SET VAR.ECCPS TO %CND_$IF(REGISTER.RMEC1_PS EQ #WO'4066',
   'Normal', %CNV_$DECIMAL_P(REGISTER.RMEC1_PS, 6)) ;
```

```
ELSE
    SET VAR.ECCPS TO 'Outside of legal range' ;
END_IF ;
!
! If the error was a non-correctable hard error or Error Correction
! was inhibited, then the ECC position and ECC pattern are irrelevant.
!
IF (REGISTER.RMER1_ECH EQ TRUE)
THEN

    BEGIN
    SET VAR.ECCPS  TO 'Irrelevant (ECH set)' ;
    SET VAR.ECCPAT TO 'Irrelevant (ECH set)' ;
    END ;
END_IF ;

IF (REGISTER.RMOF [11] EQ TRUE)
THEN

    BEGIN
    SET VAR.ECCPS  TO 'Irrelevant (ECI set)' ;
    SET VAR.ECCPAT TO 'Irrelevant (ECI set)' ;
    END ;

ELSE
    SET VAR.ECCPAT TO %CNV_$OCTAL(REGISTER.RMEC2 [0:11], 4, '0') | ' (0)' ;
END_IF ;
                                        ❻
!
! The following will use the register information to determine the
! value of the  intermodule variables,  which are needed  by  the
! dispatcher and stuff these accordingly.
!
! The variables are:
!
!   INTERMOD_DEVERR.DRIVE_SN
!   INTERMOD_DEVERR.DEV_FUNCTION
!   INTERMOD_DEVERR.PHYS_UNIT
!   INTERMOD_DEVERR.ERROR_CYLINDER
!   INTERMOD_DEVERR.ERROR_SECTOR
!   INTERMOD_DEVERR.ERROR_HEAD
!   INTERMOD_DEVERR.ERROR_GROUP   (not applicable to this device)
!   INTERMOD_DEVERR.BLOCK_NUMBER
!   INTERMOD_DEVERR.ERROR_TYPE
!
! Return the drive serial number.
!
SET INTERMOD_DEVERR.DRIVE_SN TO %CNV_$BCD(REGISTER.RMSN, 12, ' ') ;
```

```
!
! Look up the function code in the function table.
!
FIND FUNCTION FUN_CODE = REGISTER.RMCS1_FN ;

!
! Check if a match is found between the register and the table,
!
IF FUNCTION.CONTEXT

THEN
    !
    ! Yes, return the associated function text in the variable.
    !
    SET INTERMOD_DEVERR.DEV_FUNCTION TO FUNCTION.FUN_TEXT ;

ELSE
    !
    ! Otherwise, return text indicating an invalid function.
    !
    SET INTERMOD_DEVERR.DEV_FUNCTION TO 'Invalid function' ;

END_IF ;

!
! Return the physical unit number.
!
SET INTERMOD_DEVERR.PHYS_UNIT TO %CNV_$DECIMAL(REGISTER.RMCS2_UN, 1) ;

!
!      DISK GEOMETRY INFORMATION.
!

! Calculate the intermodule variables for  LBN,  GROUP,  CYLINDER, TRACK,
! and SECTOR address, initially assuming the error packet was NOT caused
! by a data error.

!
! Calculate LBN using the formula...
!
!    LBN = ( CYLINDER_ADRS * number of SECTORS/CYL +
!       HEAD_ADRS * number of SECTORS/TRACK +
!      SECTOR_ADRS )

!
SET INTERMOD_DEVERR.BLOCK_NUMBER TO
        %CNV_$DECIMAL_P(
         (REGISTER.RMDC_DC * #LD'160' +
          REGISTER.RMDA_HD  * #WD'32'  +
          REGISTER.RMDA_SEC ),
          9) ;
!

! Initialize GROUP. (not applicable to this device)
!
SET INTERMOD_DEVERR.ERROR_GROUP TO 'N/A' ;

!
! Initialize CYLINDER.
!
SET INTERMOD_DEVERR.ERROR_CYLINDER TO
    %CNV_$DECIMAL_P(REGISTER.RMDC_DC, 3) ;
```

```
!
! Initialize TRACK (head).
!
SET INTERMOD_DEVERR.ERROR_HEAD TO
    %CNV_$DECIMAL_P(REGISTER.RMDA_HD, 2) ;

!
! Initialize SECTOR.
!
SET INTERMOD_DEVERR.ERROR_SECTOR TO
    %CNV_$DECIMAL_P(REGISTER.RMDA_SEC, 2) ;

!
! Correct the geometry information if necessary.

!
! Upon a data error, the hardware will update the GROUP, CYLINDER, TRACK and
! SECTOR to point to the sector following the sector in error.   In order to
! make the intermodule variables for GROUP, CYLINDER, TRACK, SECTOR and  LBN
! point to the media address causing a data error, they are corrected (backed
! off by 1) using the following algorithm.

!
! Was it a data error ?              (check error bits)
! Yes, it was a data error.              (correction (backoff) is needed)
!     Decrement LBN.                 (recalculate pointing to previous BLK)

!     Was SECTOR = 0 ?               (sector underflow boundary?)
!         Yes, SECTOR = 0.           (underflow sector and borrow from TRK)
!          SECTOR = SECTORMAX.       (underflow the sector)

!          Was TRACK = 0?            (track underflow boundary?)
!          Yes, TRACK = 0.           (underflow TRK and borrow from CYL)
!              TRACK = TRACKMAX.     (underflow the track)
!              Decrement CYLINDER.   (borrow from CYL for TRK)

!          No, TRACK NOT = 0.        (no underflow of TRK)
!              Decrement TRACK.      (simply, with no borrow from CYL)

!          No, SECTOR NOT = 0.           (no underflow at all)
!          Decrement SECTOR.         (point to the previous block)
!  No, it was not a data error.          (no correction (backoff) needed)

!
! Was it a data error?
!
IF REGISTER.RMER1_DCK OR REGISTER.RMER1_ECH OR REGISTER.RMCS2_WC
THEN

!
! Yes, it was a data error. (LBN and geometry information needs correction)

!
BEGIN
    !
    ! Correct the LBN by recalculating (backed off by one block).
    !
    SET INTERMOD_DEVERR.BLOCK_NUMBER TO
            %CNV_$DECIMAL_P(
             (REGISTER.RMDC_DC * #LD'160' +
              REGISTER.RMDA_HD  * #WD'32'  +
```

```
            REGISTER.RMDA_SEC ) -1,
            9) ;
    !
    ! Was the sector address zero? (sector underflow?)
    !
    IF REGISTER.RMDA_SEC EQ #BD'00'
    THEN

    !
    ! Yes, it was zero. (so underflow the sector and borrow from track)
    !

    BEGIN
   !
   ! Underflow the sector.
   !
   SET INTERMOD_DEVERR.ERROR_SECTOR TO '31.' ;

   !
   ! Was track (head) address zero? (track underflow?)
   !
   IF REGISTER.RMDA_HD EQ #BD'00'
   THEN
        !
        ! Yes, the track was 0, so underflow the track
        ! and borrow from the cylinder.
        !

   BEGIN
       !
           ! Underflow the track (head).
       !
       SET INTERMOD_DEVERR.ERROR_HEAD TO '4.' ;

       !
       ! Borrow from the cylinder.
       !
       SET INTERMOD_DEVERR.ERROR_CYLINDER TO
             %CNV_$DECIMAL_P(REGISTER.RMDC_DC - 1, 3) ;
   END ;

       ELSE
       !
       ! No, the track was not zero. Simply decrement it. (no track underflow)
       !
       SET INTERMOD_DEVERR.ERROR_HEAD TO
             %CNV_$DECIMAL_P(REGISTER.RMDA_HD - 1, 2) ;
       END_IF ;
    END ;

    ELSE
        !
        ! No, the sector address was not zero. Simply decrement it.
        ! (no sector underflow)

        !
        SET INTERMOD_DEVERR.ERROR_SECTOR TO
            %CNV_$DECIMAL_P(REGISTER.RMDA_SEC - 1, 2) ;
    END_IF ;
END ;
END_IF ;
```

❼

```
!
! Find the reason causing this error packet and set the variable
! accordingly.
!

IF REGISTER.RMCS1_TRE
THEN
    BEGIN
       IF NOT REGISTER.RMDS_ERR
       THEN
            DECODE
                INTERMOD_DEVERR.ERROR_TYPE = REGISTER ;

                    RMCS2 [15] ;    ! Data Late
                    RMCS2 [14] ;    ! Write Check Error
                    RMCS2 [13] ;    ! U.B. Parity Error
                    RMCS2 [12] ;    ! Nonexistent Drive
                    RMCS2 [11] ;    ! Nonexistent Memory
                    RMCS2 [10] ;    ! Program Error
                    RMCS2 [ 9] ;    ! Missed Transfer
                    RMCS2 [ 8] ;    ! MASSBUS Data Bus Parity Error
            END_DECODE ;
       ELSE
            DECODE
                INTERMOD_DEVERR.ERROR_TYPE = REGISTER ;

                    RMER2 [15] ;    ! Bad Sector Detected (Hdr bit)
                    RMER2 [14] ;    ! Seek Incomplete
                    RMER2 [13] ;    ! Operator Plug Error (removed)
                    RMER2 [12] ;    ! Invalid Command (VV bit reset)
                    RMER2 [11] ;    ! Lost of System Clock
                    RMER2 [10] ;    ! Lost of Bit Clock
                    RMER2 [ 7] ;    ! Device Check
                    RMER2 [ 3] ;    ! Data Parity Error
                    RMER1 [ 6] ;    ! ECC Hard Error
                    RMER1 [15] ;    ! Data Check
                    RMER1 [14] ;    ! Drive Unsafe
                    RMER1 [13] ;    ! Operation Incomplete
                    RMER1 [12] ;    ! Drive Timing Error
                    RMER1 [11] ;    ! Write Lock Error
                    RMER1 [10] ;    ! Invalid Address Error
                    RMER1 [ 9] ;    ! Address Overflow Error
                    RMER1 [ 8] ;    ! Header CRC Error
                    RMER1 [ 7] ;    ! Header Compare Error
                    RMER1 [ 5] ;    ! Write Clock Fail
                    RMER1 [ 4] ;    ! Format Error
                    RMER1 [ 3] ;    ! Parity Error
                    RMER1 [ 2] ;    ! Register Modification Refused
                    RMER1 [ 1] ;    ! Illegal Register
                    RMER1 [ 0] ;    ! Illegal Function

            END_DECODE ;
       END_IF ;
    END ;
ELSE
    DECODE
       INTERMOD_DEVERR.ERROR_TYPE = REGISTER ;
```

```
            NOT RMDS  [12] ;    ! Medium not Online
            NOT RMDS  [ 8] ;    ! Drive not Present
            NOT RMDS  [ 6] ;    ! Volume not Valid
                RMCS1 [13] ;    ! MASSBUS Control Bus Parity Error
            NOT RMCS1 [11] ;    ! Drive not Available
            NOT RMCS1 [ 7] ;    ! Controller not Ready
    END_DECODE ;
END_IF ;

IF (INTERMOD_DEVERR.ERROR_TYPE EQ NULL)
THEN
    SET INTERMOD_DEVERR.ERROR_TYPE TO 'No error bit found' ;
END_IF ;
```

❽

```
!
! All the intermodule variables have been stuffed,  so return to the
! coroutine caller (calling dispatch module).

!
COROUTINE ;
!
! The dispatcher  returns control to this module here, with the flag
! INTERMOD_DEVERR.PRINT_FLAG  set to either  TRUE or  FALSE.  If the
! flag is TRUE, a FULL or REGISTER report is in progress, the banner
! has been printed, and this module prints device registers (or data
! fields for packet oriented devices).  Otherwise, this  module  does
! not print anything,  and simply exits back to the dispatcher.  The
! width of the report  (80/132)  is controlled by dispatcher-defined
! format variables  REPORT.W_G_F_1  and  REPORT.W_G_F_2 based on the
! user-specified /WIDTH switch.
!
```

❾

```
IF INTERMOD_DEVERR.PRINT_FLAG
THEN
    BEGIN
       !
       ! Print the header for the Name, Value and Interpretation fields.
       !

       WRITE
          FORMAT
          '!5FCName!13FCValue!25FCInterpretation!2FL' ;
       !
       ! Print the registers according to the format variable (80/132)
       ! provided by the dispatcher.
       !

   WRITE_GROUP REGISTER
       FORMAT
           !
           ! Print format for the register name
           ! and its associated value.

           REPORT.W_G_F_1,
           !
           ! Print format for the exploded bits and fields.
```

```
            REPORT.W_G_F_2 ;
        !
        ! If there are any NOTES to be printed,  this is where the
        ! PUT of note indices is done on the note file.  When the
        ! return from this module is done,  the dispatching module
        ! examines the note file  to determine if the  note module
        ! NRM23   should be called to print the notes specified by
        ! index number.
        !

        ! If the unused bits 5 to 7 are set in the RMDA register.
        !
                                                    (10)
        IF (REGISTER.RMDA [5:3] NE #BB'0')
        THEN
            !
            ! Print the note saying that it may cause an invalid
            ! sector address to be recognized resulting in a
            ! possible invalid address error.
            !
            PUT NOTE_NUMBERS INDEX = 1 ;
         END_IF ;
    END ;
END_IF ;
END ;
                                                    (11)

END_MODULE ;
```

4.6.2 The DSP2P1 Dispatcher Module

The following example is an annotated listing of the DSP2P1 dispatcher module:

```
MODULE DSP2P1 'P01.00' ;
!
! ERROR LOG CONTROL FILE MODULE: DSP2P1
!
!      COPYRIGHT (c) 1981 BY
!   DIGITAL EQUIPMENT CORPORATION, MAYNARD
!    MASSACHUSETTS.  ALL RIGHTS RESERVED.

! THIS  SOFTWARE  IS  FURNISHED  UNDER  A LICENSE AND MAY BE USED
! AND  COPIED  ONLY IN  ACCORDANCE WITH THE TERMS OF SUCH LICENSE
! AND WITH  THE INCLUSION  OF THE ABOVE  COPYRIGHT  NOTICE.  THIS
! SOFTWARE, OR ANY OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
! OTHERWISE MADE  AVAILABLE TO ANY OTHER PERSON.  NO TITLE TO AND
! OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.

! THE INFORMATION  IN THIS DOCUMENT IS SUBJECT  TO CHANGE WITHOUT
! NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT  BY  DIGITAL
! EQUIPMENT CORPORATION.

! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
! ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
!
! VERSION 01.00

! C. PUTNAM   22-SEP-80
```

```
! This module is called to process Device Error packets.
!
!   Module Name:            DSP 2 P 1
!                           --- - - -
!                            ^  ^ ^ ^
!   Module Prefix: ----------!  ! ! !
!                              ! ! !
!   Error Code: ---------------! ! !
!                                ! !
!   Operating System: ------------! !
!                                  !
!   Packet Format: -----------------!

! The following Error Subcodes are defined:
!
!   Subcode      Mnemonic   Meaning
!   -------      --------   -------
!
!      1         E_$SDVH      Device Hard Error
!      2         E_$SDVS      Device Soft Error
!      3         E_$STMO      Device Interrupt Timeout
!      4         E_$SUNS      Unsolicited Interrupt
!

! Define any literals used in this module.
!
LITERAL DSP2_SUB_ANY.FORMAT_1 =
    'I/O Operation Information:!1FL' |
    '--- --------- ------------!2FL' |
    '!5FCDevice Function!38FCType of Error!2FL' |
    '!5FC!30DP!38FC!30DP!3FL' ;

LITERAL DSP2_SUB_ANY.FORMAT_2 =
    'Device Error Position Information:!1FL' |
    '------ ----- -------- ------------!2FL' |
    '!5FCCylinder!15FCGroup!22FCHead!28FCSector!36FCBlock!2FL' |
    '!5FC!8DP!15FC!5DP!22FC!4DP!28FC!6DP!36FC!10DP!2FL' ;

PROCEDURE START_MOD
BEGIN
!
! Create the Subcode Conversion table.
!
TABLE SUBCODE ;
    NUMBER    :WORD ;
    TEXT      :ASCII [18] ;

BEGIN_TABLE
    1,         'Device Hard Error' ;
    2,         'Device Soft Error' ;

    3,         'Device Timeout' ;
    4,         'Spurious Interrupt' ;
END_TABLE ;
```

❶

```
! Create a flag that indicates whether the device is a magtape or not.

DECLARE INDICATE ;
    TAPE_FLAG      :LOGICAL ;
END_DECLARE ;
```

```
! First check to see if PERIPHERAL errors are selected. If they are not,
! simply return. Also determine the packet subtype. If it is a known
! subtype code, then proceed. Otherwise it is an error.

IF NOT REPORT.PERIPHERAL

THEN
    !
    ! This type of packet has not been selected for printing.
    !
    RETURN ;

END_IF ;

FIND SUBCODE NUMBER = HEADER.CODE_SUBTYPE ;

IF NOT SUBCODE.CONTEXT

THEN

    BEGIN
    SIGNAL 'ILLPACSBC' PARAMETERS
       REPORT.PACKET_IDENT,
       %CNV_$DECIMAL(HEADER.CODE_TYPE, 3),
       %CNV_$DECIMAL(HEADER.CODE_SUBTYPE, 3) ;

    RETURN ;
    END ;

END_IF ;
```

❷

```
!
! Now find the device name by calling the DEVICE_NAME procedure.
!
CALL MODULE 'DEVSM1' PROCEDURE 'DEVICE_NAME' ;
```

❸

```
!
! Prepare the NOTE_NUMBERS file for any notes that may be requested.
!
POINTER NOTE_NUMBERS CLEAR ;
```

❹

```
!
! Now set up the procedure DEVICE_ERROR and the appropriate device-level
! module as a coroutine pair.
!
CALL MODULE INTERMOD_DEVERR.DISP_NAME PROCEDURE 'DEVICE_ENTRY'
     COROUTINE 'DEVICE_ERROR' ;
END ;

PROCEDURE DEVICE_ERROR
BEGIN
!
! The following is used to format the output for the
! 'Device Hard Error', 'Device Soft Error', 'Device Interrupt Timeout'
! and 'Spurious Interrupt' Device Error packets.

! The DEVICE_ID subpacket contains information about the
! device on which the error occurred.

! The IO_ACTIVITY subpacket contains information about all
! other concurrent I/O in the system.
```

```
! The DEVICE_OP subpacket contains information about the I/O
! Operation in progress on the device at the time of the error.
!
                                                    ❺
! Obtain information from the coroutine partner.
!
COROUTINE ;
                                                    ❻
!
! Assume the serial number test will succeed or be irrelevant.
!
SET INTERMOD_DEVERR.REJECT_FLAG TO FALSE ;

! Now test to see if this device passes the drive serial number test.
!
IF REPORT.DRIVE_SN_VALID AND
    (INTERMOD_DEVERR.DRIVE_SN NE %CNV_$BCD(REPORT.DRIVE_SN, 12))

THEN
    !
    ! Indicate that the test failed.
    !
    SET INTERMOD_DEVERR.REJECT_FLAG TO TRUE ;

END_IF ;

!
! Determine the type of report and format the output
! accordingly.

CASE REPORT.MODE OF

    ['BRIEF']:
       !
       ! The BRIEF report is one line long.
       !
       BEGIN
       !
       ! Now output the information based on the result of the test.
                                                    ❼
   IF NOT INTERMOD_DEVERR.REJECT_FLAG

   THEN
       !
       ! Now output the brief report.

       BEGIN
       WRITE
                REPORT.PACKET_IDENT,
                %CNV_$RSX_TIME(HEADER.TIME_STAMP, 0),
                SUBCODE.TEXT,
                DISP.DEVICE_STRING,
                INTERMOD_DEVERR.ERROR_TYPE,
                'Function = ' | INTERMOD_DEVERR.DEV_FUNCTION
            FORMAT
                REPORT.BRIEF_FORMAT ;
```

```
        ! Now increment the printed packet count.
        !
        INCREMENT REPORT.PRINT_COUNT ;
        END ;

    END_IF ;

    !
    ! Now go back to the partner. It will simply return
    ! without printing.
    !
    SET INTERMOD_DEVERR.PRINT_FLAG TO FALSE ;
    END ;

['FULL', 'REGISTERS']:
    !
    ! The FULL report contains detailed information
    ! about the error.
    !

    BEGIN
    !
    ! Now output the information based on the result of the test.
                                          ❽
    !
    IF NOT INTERMOD_DEVERR.REJECT_FLAG

    THEN
        !
        ! Output the first page if the report type is 'FULL'.

        BEGIN
        IF REPORT.MODE EQ 'FULL'

        THEN
             !
             ! Now output the information for the standard subpackets.
             !
             BEGIN
             CALL MODULE REPORT.FULL_MOD PROCEDURE 'OUTPUT_PACKETS' ;
             !
             ! Now output the Data subpacket information.

             WRITE
                     INTERMOD_DEVERR.DEV_FUNCTION,
                     INTERMOD_DEVERR.ERROR_TYPE
                 FORMAT
                     DSP2_SUB_ANY.FORMAT_1 ;
             !
             ! Now output the Device Error Position information
             ! if it is applicable.
             !
             IF INTERMOD_DEVERR.ERROR_CYLINDER NE '???'
```

```
    THEN
        WRITE
                INTERMOD_DEVERR.ERROR_CYLINDER,
                INTERMOD_DEVERR.ERROR_GROUP,
                INTERMOD_DEVERR.ERROR_HEAD,
                INTERMOD_DEVERR.ERROR_SECTOR,
                INTERMOD_DEVERR.BLOCK_NUMBER
            FORMAT
                DSP2_SUB_ANY.FORMAT_2 ;

    END_IF ;
    !
    ! A full report is wanted, so print the record I.D.
    ! and header.

    WRITE
            REPORT.PACKET_IDENT
        FORMAT
            '!1FP!5FCEntry !DP!22FC(continued)!3FL' |
            'Device Supplied Information:!FL' |
            '------ -------- ------------!2FL' ;
    END ;

ELSE
    !
    ! Only a register dump is requested, therefore print
    ! the banner from the full report.

    WRITE
            REPORT.PACKET_IDENT,
            %CNV_$DECIMAL_P(HEADER.ERROR_SEQ, 8),
            DISP.DEVICE_STRING,
            SUBCODE.TEXT,
            %CND_$IF((INTERMOD_DEVERR.ERROR_TYPE NE NULL),
                ' (' | INTERMOD_DEVERR.ERROR_TYPE | ')',
                NULL),
            %CNV_$RSX_TIME(HEADER.TIME_STAMP, 0)
        FORMAT

            ! Select the format statement
            ! based on the desired width.
            !
            %CND_$IF(REPORT.WIDE,
            !
            ! WIDE is selected.

            '!1FP!5FCEntry !7DP!20FCSequence !9DP' |
            '!40FC!6DP!48FC!18DP!DP!2FS!20DP!3FL' |
            'Device Supplied Information:!FL' |
            '------ -------- ------------!2FL',
            !
            ! NARROW is selected.
            !
            '!1FP!5FCEntry !7DP!20FCSequence !9DP!40FC!6DP!FL' |
            '!5FC!18DP!DP!2FS!20DP!3FL' |
            'Device Supplied Information:!FL' |
            '------ -------- ------------!2FL') ;

END_IF ;
```

```
        ! Now increment the printed packet count and tell the device
        ! module we want it to print.
        !
        INCREMENT REPORT.PRINT_COUNT ;

        SET INTERMOD_DEVERR.PRINT_FLAG TO TRUE ;
        END ;

    ELSE
        !
        ! We don't want to print because the packet was rejected.
        !
        SET INTERMOD_DEVERR.PRINT_FLAG TO FALSE ;

    END_IF ;

    ! Now go back to the partner. It will output the device registers
    ! if the print flag is true.
    !
    END ;
```

❾

```
['NONE']:
    !
    ! If the report type is NONE, output nothing.
    !
    SET INTERMOD_DEVERR.PRINT_FLAG TO FALSE ;

END_CASE ;
```

❿

```
!
! Now COROUTINE back to the partner. It will print if instructed to do so.
!
COROUTINE ;
```

⓫

```
! Test to see if the packet was accepted. If it was, update the files.
!
IF NOT INTERMOD_DEVERR.REJECT_FLAG THEN

    !
    ! Update the files.
```

⓬

```
    !
    CALL PROCEDURE 'UPDATE_RECORD' ;

END_IF ;

! Now see if any notes were requested and print them if there were.
!
IF NOTE_NUMBERS.CONTEXT THEN

    BEGIN
    SET INTERMOD_DEVERR.DISP_NAME TO 'N' |
        %STR_$REMAINING(INTERMOD_DEVERR.DISP_NAME, 2) ;

    IF %PKT_$MODULE(INTERMOD_DEVERR.DISP_NAME)

    THEN

        CALL MODULE INTERMOD_DEVERR.DISP_NAME PROCEDURE 'NOTES' ;
```

```
    ELSE
       SIGNAL 'NONOTES' PARAMETERS INTERMOD_DEVERR.DRIVE_TYPE ;
    END_IF ;
    END ;
END_IF ;
END ;

PROCEDURE UPDATE_RECORD
BEGIN
! This procedure is used to update an error type record if a record for the
! type of error exists. If it does not exist, a record is created.

! First see if a record exists in the ERROR_INFO_E file that matches
! on the following keys:
!
!      Device name
!      Device type
!      Pack SN
!      Drive SN
!      Volume label
!      Error type
                                          (13)

IF REPORT.ERROR

THEN
    !
    ! This type of summary is desired.
    !
    BEGIN
    POINTER ERROR_INFO_E FIRST ;
    !
    ! Now try to find a record that matches on all of the keys.
    !
    FIND ERROR_INFO_E
       NAME = DISP.DEVICE_STRING,
       DEVICE_TYPE = %CND_$IF(INTERMOD_DEVERR.MISMATCH_FLAG, '*', NULL) |
                          INTERMOD_DEVERR.DRIVE_TYPE,
       PACK_SN = DEVICE_ID.PACK_SN,
       DRIVE_SN = INTERMOD_DEVERR.DRIVE_SN,
       VOLUME_LABEL = DEVICE_ID.VOLUME_LABEL,
       ERROR_TYPE = INTERMOD_DEVERR.ERROR_TYPE ;
    ! See if there was a match.
    !
    IF ERROR_INFO_E.CONTEXT
    THEN
       !
       ! There was a match. Update the record to
       ! show that this error occurred.
       BEGIN
       INCREMENT ERROR_INFO_E.ERROR_COUNT ;
       IF DISP.PACKET_DATE LT ERROR_INFO_E.FIRST_DATE
       THEN
```

```
            BEGIN
            SET ERROR_INFO_E.FIRST_DATE TO DISP.PACKET_DATE ;
            SET ERROR_INFO_E.FIRST_PACKET TO REPORT.PACKET_IDENT ;
            END ;
        END_IF ;
        IF DISP.PACKET_DATE GT ERROR_INFO_E.LAST_DATE
        THEN
            BEGIN
            SET ERROR_INFO_E.LAST_DATE TO DISP.PACKET_DATE ;
            SET ERROR_INFO_E.LAST_PACKET TO REPORT.PACKET_IDENT ;
            END ;
        END_IF ;
        END ;
        ELSE
            !
            ! This is the first error of this kind. Create a record in the
            ! ERROR_INFO_E file that describes this error.
            PUT ERROR_INFO_E
                NAME = DISP.DEVICE_STRING,
                DEVICE_TYPE = %CND_$IF(INTERMOD_DEVERR.MISMATCH_FLAG, '*', NULL) |
                              INTERMOD_DEVERR.DRIVE_TYPE,
                PACK_SN = DEVICE_ID.PACK_SN,
                DRIVE_SN = INTERMOD_DEVERR.DRIVE_SN,
                VOLUME_LABEL = DEVICE_ID.VOLUME_LABEL,
                ERROR_TYPE = INTERMOD_DEVERR.ERROR_TYPE,
                ERROR_COUNT = 1,
                FIRST_DATE = DISP.PACKET_DATE,
                LAST_DATE = DISP.PACKET_DATE,
                FIRST_PACKET = REPORT.PACKET_IDENT,
                LAST_PACKET = REPORT.PACKET_IDENT ;
        END_IF ;
        END ;
END_IF ;
! First see if a record exists in the ERROR_INFO_G file that matches
! on the following keys:
!
!       Device name
!       Device type
!       Pack SN
!       Drive SN
!       Volume label
!       Block number
```

⓮

```
IF REPORT.GEOMETRY AND NOT INDICATE.TAPE_FLAG
THEN
    !
    ! This type of summary is desired.
    BEGIN
    POINTER ERROR_INFO_G FIRST ;
```

```
    ! Now try to find a record that matches on all of the keys.
    !
    FIND ERROR_INFO_G
        NAME = DISP.DEVICE_STRING,
        DEVICE_TYPE = %CND_$IF(INTERMOD_DEVERR.MISMATCH_FLAG, '*', NULL) |
                          INTERMOD_DEVERR.DRIVE_TYPE,
       PACK_SN = DEVICE_ID.PACK_SN,
       DRIVE_SN = INTERMOD_DEVERR.DRIVE_SN,
       VOLUME_LABEL = DEVICE_ID.VOLUME_LABEL,
       BLOCK_NUMBER = INTERMOD_DEVERR.BLOCK_NUMBER ;

    ! See if there was a match.
    !
    IF ERROR_INFO_G.CONTEXT

    THEN
        !
        ! There was a match. Update the record to
        ! show that this error occurred.
        !
        INCREMENT ERROR_INFO_G.ERROR_COUNT ;

    ELSE
        !
        ! This is the first error of this kind. Create a record in the
        ! ERROR_INFO_G file that describes this error.

        PUT ERROR_INFO_G
            NAME = DISP.DEVICE_STRING,
            DEVICE_TYPE = %CND_$IF(INTERMOD_DEVERR.MISMATCH_FLAG, '*', NULL) |
                              INTERMOD_DEVERR.DRIVE_TYPE,
            PACK_SN = DEVICE_ID.PACK_SN,
            DRIVE_SN = INTERMOD_DEVERR.DRIVE_SN,
            VOLUME_LABEL = DEVICE_ID.VOLUME_LABEL,
            ERROR_HEAD = INTERMOD_DEVERR.ERROR_HEAD,
            ERROR_GROUP = INTERMOD_DEVERR.ERROR_GROUP,
            ERROR_CYLINDER = INTERMOD_DEVERR.ERROR_CYLINDER,
            ERROR_SECTOR = INTERMOD_DEVERR.ERROR_SECTOR,
            BLOCK_NUMBER = INTERMOD_DEVERR.BLOCK_NUMBER,
            ERROR_COUNT = 1 ;

    END_IF ;
    END ;

END_IF ;
END ;

END_MODULE ;    ! DSP2P1.CNF
```

4.6.3 The RM02/03 Notes Module NRM23

The following example is an annotated listing of the notes module for the RM02 and RM03 disk drives:

❶

```
MODULE NRM23 'M01.00' ;
!
! ERROR LOG CONTROL FILE MODULE:  RM02, RM03 Notes
!
!      COPYRIGHT (c) 1981 BY
!   DIGITAL EQUIPMENT CORPORATION, MAYNARD
!    MASSACHUSETTS.  ALL RIGHTS RESERVED.
!
! THIS  SOFTWARE  IS  FURNISHED  UNDER  A LICENSE AND MAY BE USED
! AND  COPIED  ONLY IN  ACCORDANCE WITH THE TERMS OF SUCH LICENSE
! AND WITH  THE INCLUSION  OF THE ABOVE  COPYRIGHT  NOTICE.  THIS
! SOFTWARE,  OR ANY OTHER  COPIES  THEREOF, MAY NOT BE PROVIDED OR
! OTHERWISE MADE  AVAILABLE TO ANY OTHER PERSON.  NO TITLE TO AND
! OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.

! THE INFORMATION  IN THIS DOCUMENT IS SUBJECT  TO CHANGE WITHOUT
! NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT  BY  DIGITAL
! EQUIPMENT CORPORATION.

! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
! ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.

! VERSION 01.00
!
! R. Ryan   30-Jun-81

! This is one of the many device modules, which is called by the device
! error dispatcher (DSP2P1) or device information dispatcher (DSP3P1)
! to process notes for all the device-dependent information.
```

❷

```
PROCEDURE NOTES
!
! This procedure,  which is called from the DSP2P1 module, processes any
! requests for notes.

BEGIN
!
! Print the NOTE header and define the format for the NOTE section.
```

❸

```
WRITE FORMAT
    '!3FLNotes on RM02, RM03 errors:!2FL' ;

POINTER NOTE_NUMBERS FIRST ;

WHILE NOTE_NUMBERS.CONTEXT DO
```

❹

```
    BEGIN
    SELECT NOTE_NUMBERS._INDEX OF
       !
       ! Note number 1.
```

```
        BEGIN
        WRITE FORMAT
            '*      RMDA bits 5,6,7 are unused, however if they are' ;
        WRITE FORMAT
            '       set, they will be interpreted as the high order' ;
        WRITE FORMAT
            '       bits of the sector address.  This may result in' ;
        WRITE FORMAT
            '       an Invalid Address Error.!3FL' ;
        END ;

    ELSE
                                        ❺
        !
        ! This is an unknown note number.
        !
        SIGNAL 'UNKNWNNOT' PARAMETERS
            %CNV_$DECIMAL_P(NOTE_NUMBERS.INDEX, 3),
            INTERMOD_DEVERR.DRIVE_TYPE ;

    END_SELECT ;
                                        ❻
    POINTER NOTE_NUMBERS NEXT ;
    END ;

END ;
                                        ❼
END_MODULE ;    ! NRM23.CNF
```

4.6.4 Subpacket Definitions

This section lists the DIGITAL-standard subpackets. These subpackets are declared by the following modules:

- The HEADER subpacket contains information largely from SYSCM. It describes the characteristics of the system and packet. This subpacket is always required.

```
SUBPACKET HEADER = DISP.NEXT_PACKET ;
    SUBPKT_FLAGS        :WORD ;                 ! E$HSBF
        SFLG_HDR        :BIT [0] ;              ! HEADER Subpacket
        SFLG_TSK        :BIT [1] ;              ! TASK Subpacket
        SFLG_DID        :BIT [2] ;              ! DEVICE_ID Subpacket
        SFLG_DOP        :BIT [3] ;              ! DEVICE_OP Subpacket
        SFLG_DAC        :BIT [4] ;              ! DEVICE_AC Subpacket
        SFLG_DAT        :BIT [5] ;              ! DATA Subpacket
        SFLG_MBC        :BIT [13] ;             ! 22-bit MASSBUS Controller
        SFLG_CMD        :BIT [14] ;             ! Command Subpacket
        SFLG_ZER        :BIT [15] ;             ! I/O counts zeroed

    OP_SYS              :BYTE ;                 ! E$HSYS
    FORMAT_ID           :BYTE ;                 ! E$HIDN
    OP_SYS_ID           :ASCII [4] ;            ! E$HSID
```

```
    CONTEXT_CODE        :BYTE ;                 ! E$HCTX
        CC_NOR          :BIT [0] ;              ! Normal Entry
        CC_STA          :BIT [1] ;              ! Start Entry
        CC_CDA          :BIT [2] ;              ! CDA Entry
    FLAGS               :BYTE ;                 ! E$HFLG
        FLG_ADR         :FIELD [0:2] ;          ! Addressing mode
        FLG_COU         :BIT [2] ;              ! Error Counts supplied
        FLG_QBS         :BIT [3] ;              ! Q-BUS system

    ENTRY_SEQ           :WORD ;                 ! E$HENS
    ERROR_SEQ           :WORD ;                 ! E$HERS
    CODE_TYPE           :BYTE ;                 ! E$HTYC
    CODE_SUBTYPE        :BYTE ;                 ! E$HTYS
    TIME_STAMP          :RSX_TIME ;             ! E$HTIM
    PROC_TYPE           :BYTE ;                 ! E$HPTY
    RESERVED            :BYTE ;                 ! Reserved byte
    PROC_ID             :WORD ;                 ! E$HURM
        URM_CPU         :FIELD [0:4] ;          ! Processor Identifier
END_PACKET ;
```

- The TASK subpacket contains information about either the task that logged the packet or the task that caused the packet to be logged.

```
SUBPACKET TASK = DISP.NEXT_PACKET ;
    TASK_NAME           :LONGWORD ;             ! E$TTSK
    UIC                 :WORD ;                 ! E$TUIC
        UIC_MEMBER      :FIELD [0:8] ;          ! Member number in UIC
        UIC_GROUP       :FIELD [8:8] ;          ! Group number in UIC
    TI_DEV              :ASCII [2] ;            ! E$TTID
    TI_UNIT             :BYTE ;                 ! E$TTIU
    FLAGS               :BYTE ;                 ! E$TFLG
        FLG_PRV         :BIT [0] ;              ! Privileged Task
        FLG_PRI         :BIT [1] ;              ! Privileged Terminal
```

- The DEVICE_ID subpacket contains information about the device on which the error occurred.

```
SUBPACKET DEVICE_ID = DISP.NEXT_PACKET ;
     MNEMONIC           :ASCII [2] ;            ! E$ILDV
     LOGICAL_UNIT       :BYTE ;                 ! E$ILUN
     CONTROLLER_NUM     :BYTE ;                 ! E$IPCO
     PHYS_UNIT          :BYTE ;                 ! E$IPUN
     PHYS_SUB_UNIT      :BYTE ;                 ! E$IPSU

     $IF SUPPORT.RSX_11M_PLUS

     $THEN

         IF OP_SYS.SUFFIX EQ 'P'

         THEN

     BEGIN
     PHYS_DEV_MNEMON    :ASCII [2] ;            ! E$IPDV
     END ;

         END_IF ;

     $END_IF

     DEV_FLAGS          :BYTE ;                 ! E$IFLG
         DFLG_SUB       :BIT [0] ;              ! Subcontroller device
```

```
    $IF SUPPORT.RSX_11M_PLUS

    $THEN
        DFLG_NUX        :BIT [1] ;              ! No UCB extension
    $END_IF
    RESERVED            :BYTE ;                 ! Reserved byte
    VOLUME_LABEL        :ASCII [12] ;           ! E$IVOL
    PACK_SN             :LONGWORD ;             ! E$IPAK
    DEV_TYPE_CLASS      :WORD ;                 ! E$IDCL
    DEV_TYPE            :LONGWORD ;             ! E$IDTY
    IO_COUNT            :LONGWORD ;             ! E$IOPR
    SOFT_ERCNT          :BYTE ;                 ! E$IERS
    HARD_ERCNT          :BYTE ;                 ! E$IERH

    $IF SUPPORT.RSX_11M_PLUS

    $THEN
        IF OP_SYS.SUFFIX EQ 'P'
        THEN
    BEGIN
    WRD_XFR_COUNT       :LONGWORD ;             ! E$IBLK
    CYL_CRS_COUNT       :LONGWORD ;             ! E$ICYL
    END ;
        END_IF ;
    $END_IF
END_PACKET ;
```

- The DEVICE_OP subpacket contains information about the requested I/O operation.

```
SUBPACKET DEVICE_OP = DISP.NEXT_PACKET ;
    TASK_NAME           :LONGWORD ;             ! E$OTSK
    UIC                 :WORD ;                 ! E$OUIC
        UIC_MEMBER      :FIELD [0:8] ;          ! Member number in UIC
        UIC_GROUP       :FIELD [8:8] ;          ! Group number in UIC
    TI_DEV              :ASCII [2] ;            ! E$OTID
    TI_UNIT             :BYTE ;                 ! E$OTIU
    RESERVED            :BYTE ;                 ! Reserved Byte
    IO_FUNCTION         :WORD ;                 ! E$OFNC
        SF_IQX          :BIT [0] ;              ! IQ.X subfunction bit
        SF_IQQ          :BIT [1] ;              ! IQ.Q subfunction bit
        SF_IQUMD        :BIT [2] ;              ! IQ.UMD subfunction bit
    FLAGS               :BYTE ;                 ! E$OFLG
        FLG_TRA         :BIT [0] ;              ! Transfer operation
        FLG_DMA         :BIT [1] ;              ! DMA device
        FLG_BAE         :BIT [2] ;              ! 22-bit addressing device
    RESERVED            :BYTE ;                 ! Reserved Byte
    XFER_ADDRESS_1      :WORD ;                 ! E$OADD + 0
        XFER1_HIGH_2    :FIELD [4:2] ;          ! High Order 2 bits of address
        XFER1_HIGH_6    :FIELD [0:6] ;          ! High Order 6 bits of address
    XFER_ADDRESS_2      :WORD ;                 ! E$OADD + 2
        XFER2_TAUB      :FIELD [0:13] ;         ! T. A. in units of bytes
    XFER_BYTE_COUNT     :WORD ;                 ! E$OSIZ
    RETRIES_LEFT        :BYTE ;                 ! E$ORTY
    MAX_RETRIES         :BYTE ;                 ! E$ORTY+1
END_PACKET ;
```

- The IO_ACTIVITY subpacket contains information about other I/O occurring in the system at the time the error was detected.

```
SUBPACKET IO_ACTIVITY = DISP.NEXT_PACKET REPEATED ;
    MNEMONIC            :ASCII [2] ;          ! E$ALDV
    LOGICAL_UNIT        :BYTE ;               ! E$ALUN
    CONTROLLER_NU       :BYTE ;               ! E$APCO
    PHYS_UNIT           :BYTE ;               ! E$APUN
    PHYS_SUB_UNIT       :BYTE ;               ! E$APSU

    $IF SUPPORT.RSX_11M_PLUS

    $THEN

        IF OP_SYS.SUFFIX EQ 'P'

        THEN

    BEGIN
    PHYS_DEV_MNEMON     :ASCII [2] ;          ! E$IPDV
    END ;

        END_IF ;

    $END_IF

    DEV_FLAGS           :BYTE ;               ! E$IFLG
        DFLG_SUB        :BIT [0] ;            ! Subcontroller device
    $IF SUPPORT.RSX_11M_PLUS

    $THEN

        DFLG_NUX        :BIT [1] ;            ! No UCB extension

    $END_IF

    TI_UNIT             :BYTE ;               ! E$ATIU
    TASK_NAME           :LONGWORD ;           ! E$ATSK
    UIC                 :WORD ;               ! E$AUIC
        UIC_MEMBER      :FIELD [0:8] ;        ! Member number in UIC
        UIC_GROUP       :FIELD [8:8] ;        ! Group number in UIC
    TI_DEV              :ASCII [2] ;          ! E$ATID
    IO_FUNCTION         :WORD ;               ! E$AFNC
        SF_IQX          :BIT [0] ;            ! IQ.X subfunction bit
        SF_IQQ          :BIT [1] ;            ! IQ.Q subfunction bit
        SF_IQUMD        :BIT [2] ;            ! IQ.UMD subfunction bit
    FLAGS               :BYTE ;               ! E$AFLG
        FLG_TRA         :BIT [0] ;            ! Transfer operation
        FLG_DMA         :BIT [1] ;            ! DMA device
        FLG_BAE         :BIT [2] ;            ! 22-bit addressing device
    RESERVED            :BYTE ;               ! Reserved Byte
    XFER_ADDRESS_1      :WORD ;               ! E$AADD + 0
        XFER1_HIGH_2    :FIELD [4:2] ;        ! High Order 2 bits of address
        XFER1_HIGH_6    :FIELD [0:6] ;        ! High Order 6 bits of address
    XFER_ADDRESS_2      :WORD ;               ! E$AADD + 2
        XFER2_TAUB      :FIELD [0:13] ;       ! T. A. in units of bytes
    XFER_BYTE_COUNT     :WORD ;               ! E$ASIZ
END_PACKET ;
```

4.6.4.1 Subpackets Declared by DSP1P1

This section describes the DATA subpackets declared by DSP1P1.

- The following DATA subpacket (Code = 1, Subcode = 1) contains information about a status-change operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    LIMIT_CODE          :BYTE ;
    LOG_CODE            :BYTE ;
    FLAGS               :BYTE ;
        FLG_CRE         :BIT [0] ;
    FILE_SPEC_LEN       :BYTE ;
    FILE_SPEC           :ASCII [80] ;
END_PACKET ;
```

- The following DATA subpacket (Code = 1, Subcode = 2) contains information about a switch-logging-files operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    RESERVED            :WORD ;
    FLAGS               :BYTE ;
        FLG_CRE         :BIT [0] ;
        FLG_DEL         :BIT [1] ;
    FILE_SPEC_LE        :BYTE ;
    FILE_SPEC           :ASCII [80] ;
END_PACKET ;
```

- The following DATA subpacket (Code = 1, Subcode = 3) contains information about an append-to-file operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    RESERVED            :WORD ;
    FLAGS               :BYTE ;
        FLG_CRE         :BIT [0] ;
        FLG_DEL         :BIT [1] ;
    FILE_SPEC_LEN       :BYTE ;
    FILE_SPEC           :ASCII [80] ;
END_PACKET ;
```

- The following DATA subpacket (Code = 1, Subcode = 4) contains information about a set-backup-file operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    RESERVED            :WORD ;
    FLAGS               :BYTE ;
    FILE_SPEC_LEN       :BYTE ;
    FILE_SPEC           :ASCII [80] ;
END_PACKET ;
```

- The following DATA subpacket (Code = 1, Subcode = 6) contains information about a change-limits operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET REPEATED ;
    HARD_LIM_FLAG       :BYTE ;
        NEW_LIMH        :BIT [0] ;
    HARD_LIMIT          :BYTE ;
    SOFT_LIM_FLAG       :BYTE ;
        NEW_LIMS        :BIT [0] ;
    SOFT_LIMIT          :BYTE ;
    MNEMONIC            :ASCII [2] ;
    LOGICAL_UNIT        :BYTE ;
    RESERVED            :BYTE ;
END_PACKET ;
```

4.6.4.2 Subpackets Declared by DSP2P1

The DATA subpackets for device errors (Code = 2, Subcodes = 1, 2, 3) contain information that is specific to each device. See the appropriate device-level module for the format of the DATA subpacket.

4.6.4.3 Subpackets Declared by DSP3P1

The DATA subpackets for device information messages (Code = 3, Subcode = 1) contain information that is specific to each device. See the appropriate device-level module for the format of the DATA subpacket.

4.6.4.4 Subpackets Declared by DSP4P1

There is no DATA subpacket for mount, dismount, and device-information reset events (Code = 4, Subcodes = 1, 2, 3).

The following DATA subpacket (Code = 4, Subcode = 4) contains information about a block-replacement operation:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    FLAGS               :WORD ;
        PRIMARY_RBN     :BIT [0] ;
        SUCCESS         :BIT [1] ;
    LBN                 :LONGWORD ;
    NEW_RBN             :LONGWORD ;
    OLD_RBN             :LONGWORD ;
END_PACKET ;
```

4.6.4.5 Subpackets Declared by DSP5P1

The following DATA subpacket (Code = 5, Subcode = 1) contains information about a memory parity error event:

```
SUBPACKET REGISTER = DISP.NEXT_PACKET NAMED ;
    RESERVED            :WORD ;                 ! Mask for Cache Registers
    RESERVED            :WORD ;                 ! Mask for Parity CSRs
    P_CSR00             :WORD ;                 ! Memory Parity CSR 00
    P_CSR01             :WORD ;                 ! Memory Parity CSR 01
    P_CSR02             :WORD ;                 ! Memory Parity CSR 02
```

```
    P_CSR03             :WORD ;             ! Memory Parity CSR 03
    P_CSR04             :WORD ;             ! Memory Parity CSR 04
    P_CSR05             :WORD ;             ! Memory Parity CSR 05
    P_CSR06             :WORD ;             ! Memory Parity CSR 06
    P_CSR07             :WORD ;             ! Memory Parity CSR 07
    P_CSR08             :WORD ;             ! Memory Parity CSR 08
    P_CSR09             :WORD ;             ! Memory Parity CSR 09
    P_CSR10             :WORD ;             ! Memory Parity CSR 10
    P_CSR11             :WORD ;             ! Memory Parity CSR 11
    P_CSR12             :WORD ;             ! Memory Parity CSR 12
    P_CSR13             :WORD ;             ! Memory Parity CSR 13
    P_CSR14             :WORD ;             ! Memory Parity CSR 14
    P_CSR15             :WORD ;             ! Memory Parity CSR 15
    LOW_ERR             :WORD ;             ! Low Error Address Register
    HIGHERR             :WORD ;             ! High Error Address Register
    CACHERR             :WORD ;             ! Cache Error Register
    CSHCTRL             :WORD ;             ! Cache Control Register
    CSHMAIN             :WORD ;             ! Cache Maintenance Register
    CACHET              :WORD ;             ! Cache Hit/Miss Register
END_PACKET ;
```

The following DATA subpacket (Code = 5, Subcode = 2) contains information about an unknown interrupt event:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
     VECTOR_OVER_FOUR   :BYTE ;
     LOST_INT           :BYTE ;
END_PACKET ;
```

4.6.4.6 Subpackets Declared by DSP6P1

The power recovery event (Code = 6, Subcode = 1) has no DATA subpacket.

4.6.4.7 Subpackets Declared by DSP7P1

The following DATA subpacket (Code = 7, Subcode = 1) contains information about a time-change event:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    NEW_TIME            :RSX_TIME ;
END_PACKET ;
```

The following DATA subpacket (Code = 7, Subcode = 2) contains information about a system-crash event:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    CRASH_TIME          :RSX_TIME ;
    OP_SYS              :BYTE ;
    FORMAT_ID           :BYTE ;
    OP_SYS_ID           :ASCII [4] ;
    TASK_NAME           :LONGWORD ;
    TI_DEV              :ASCII [2] ;
    TI_UNIT             :BYTE ;
    FLAGS               :BYTE ;
    KERNEL_APR5         :LONGWORD ;
    URM                 :WORD ;
        URM_CPU         :FIELD [0:4] ;
END_PACKET ;
```

The following DATA subpacket (Code = 7, Subcodes = 3, 4) contains information about a driver-load or driver-unload event:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    DRIVER_NAME         :ASCII [2] ;
END_PACKET ;
```

The following DATA subpacket (Code = 7, Subcode = 6) contains information about a system-message event:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    MESSAGE_LEN         :WORD ;
    MESSAGE_TEXT        :ASCII [80] ;
END_PACKET ;
```

4.6.4.8 Subpackets Declared by DSP8P1

The following DATA subpacket (Code = 8, Subcode = 2) contains information about an unexpected interrupt:

```
SUBPACKET DATA = DISP.NEXT_PACKET ;
    VECTOR_OVER_FOUR        :BYTE ;
    LOST_INT                :BYTE ;
END_PACKET ;
```

Chapter 5
Control File Language Guide

This chapter describes the Control File Language used by the Report Generator of the error logging system.

5.1 Control File Overview

The control file for the Report Generator task (RPT) describes the format of the error log file and the format of reports based on the file. The actions specified are executed for each event to produce a report. The control file specifies the format of the data in an error log packet and the output format of the report. In addition, the control file specifies information on the accumulation of summary information, how to derive additional information, and the handling of selection criteria for reports.

Control file modules are ASCII text files containing a series of statements written in the Control File Language (CFL). The CFL compiler produces intermediate-form modules to be placed in a universal library. This library is the control file.

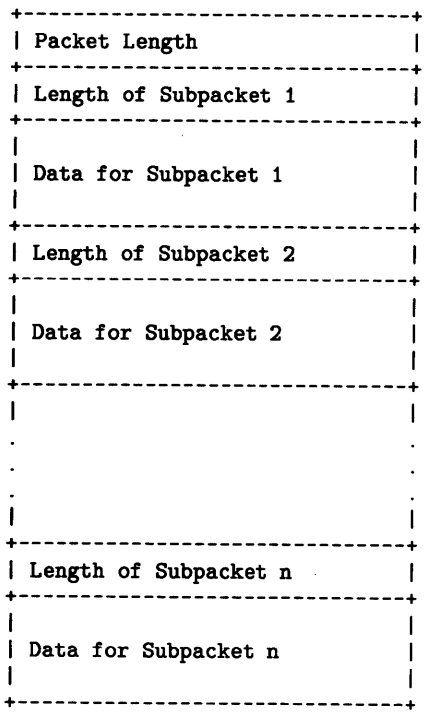
5.1.1 The Report Generator—General Processing

The Report Generator is an interpreter for the intermediate-form modules contained in the control file. RPT processes control file modules which can, in turn, process error log files. The control file module, written in CFL, specifies the processing to be performed on the current packet. The processing usually involves calling subroutines from other modules.

CFL includes primitives that stop packet processing because of an error or because the packet does not meet specified selection criteria.

5.1.2 The General Format of an Error Log Packet

The following diagram illustrates how data is organized in a typical error log packet:



Each subpacket contains a different type of data. The information in the subpackets, taken together, describes an event logged by the error logging system. The control file contains primitives for describing a subpacket so that its contents can be manipulated symbolically. Other primitives describe the entire packet as a unit.

5.1.3 The Control File Language

CFL is a specialized language designed for this application. It is a statement-oriented, block-structured language similar in concept to PASCAL and ALGOL. Unlike BLISS, CFL is not expression-oriented; it differentiates between statements and expressions.

CFL has some of the other capabilities of BLISS, however. For example, expressions can contain conditionals. This feature handles the more complex data formats of error log files. CFL does not include the full set of primitives required of an ordinary general-purpose language, but a number of specialized primitives speed up and simplify the handling of common error log data formats.

5.1.4 The General Format of Control File Modules

Each control file module contains lines of ASCII text. Each line is a sequence of elements, such as keywords, variable names, numbers, and operators. Spaces and tabs separate atomic items, such as keywords or names. Excess spaces and tabs are ignored and can be used freely for formatting.

You can insert comments in the module text by prefixing them with an exclamation point (!). The compiler ignores any text on a line following an exclamation point.

The three basic elements of CFL are statements, expressions, and declarations, as follows:

- A statement describes an action. Statements begin with a keyword and are terminated by a semicolon (;).
- An expression describes a computation. Expressions are terminated by any nonexpression keyword or by a comma (,). A nonexpression keyword is a keyword that is not valid in an expression. Expressions can also be terminated with a semicolon (;). You can also include an expression in an expression list, using conditionals to determine whether or not a given statement or expression is to be executed. Expression lists consist of a fixed number of expressions, separated by commas.
- A declaration defines the contents of packets and subpackets and defines groups of tables for evaluating packets and subpackets. The syntax of a declaration differs for each use. Group declarations start with the DECLARE string and end with the END_DECLARE string. Table declarations begin with the TABLE string and end with END_TABLE. Dynamic table declarations begin with the DYNAMIC_TABLE string and end with END_TABLE. Packet and subpacket declarations begin with the PACKET and SUBPACKET strings, respectively, and end with END_PACKET.

5.1.5 Files

CFL can obtain input from and direct output to any one of a set of files. The files have the following internal names:

INPUT
The data input file. Data packets are read from the file. The control file can open this file, close it, and read packets from it.

REPORT
The report output file. Lines of ASCII text are written to this file. The control file can open this file, close it, and write ASCII text to it. Automatic paging is available for this file.

USER
The user-prompting file. The user can be prompted for input, and input read from this file. The control file can open this file, close it, and read ASCII data from this file, with optional prompting.

COMMAND
The command input file. The user can be prompted for input, and input read from this file. The control file can read ASCII data from this file, with optional prompting.

ERROR
The error output file. The control file can write ASCII data to this file.

5.2 Types and Expressions

RPT types data to allow easy manipulation of error log information. Expressions describe data values used by RPT. This section describes the attributes of supported data types and the format of expressions.

5.2.1 Data Types

CFL supports the following eight data types: logical, string, numeric, ASCII, field, pointer, RSX_TIME, and VMS_TIME. The evaluation of an expression results in a value, which is one of the supported data types. The data type of an expression is determined from its context.

The only automatic conversions are from string values to numeric values, and conversions of numeric values between different numeric types and field types.

The following sections describe the data types in detail.

5.2.1.1 The LOGICAL Type

The LOGICAL type expresses the Boolean values true and false. A LOGICAL type is equivalent to a BIT type. No other automatic conversions are performed to or from LOGICAL types. You express the literal values for this type with the keywords TRUE and FALSE.

5.2.1.2 The STRING Type

The STRING type represents strings of binary bytes. Literal values for the STRING type cannot be represented. String operations must be used to construct string literals.

For purposes of conversion, numeric values are considered exactly equivalent to strings. (See Section 5.2.1.4.) The length of the string is the number of bytes used to represent a value of the numeric type. For example, a WORD is equivalent to a string of length 2. The following equivalences are used:

Type	Equivalent String
BYTE	String of length 1
WORD	String of length 2
LONGWORD	String of length 4
QUADWORD	String of length 8

Strings of length 4 or less are converted to numeric types by appending leading zero bytes to form a longword. Strings of length 5 to 8 are converted to numeric types by appending leading zero bytes to form a quadword. Strings greater than length 8 are not converted to numeric types.

String declaration requires that you specify the maximum size of the string. The syntax for string declaration is as follows:

```
STRING[size]
```

If the string is a variable, it can contain any number of elements up to and including the specified maximum. If the string is part of a data declaration, it contains exactly the number of characters specified.

5.2.1.3 The ASCII Type

The ASCII type represents character strings. ASCII string literals are represented by character strings enclosed in a pair of apostrophes ('string'). Two successive apostrophes in a quoted string represent a single apostrophe. Therefore, the string 'ABC''DE' represents the string literal ABC'DE. The keyword NULL represents the null string. There is no automatic conversion to or from ASCII strings. You cannot use the quotation mark (") to enclose such strings, nor does the quotation mark require flagging.

ASCII string declaration requires specification of the maximum length of the string. Specify the maximum string length as follows:

```
ASCII[size]
```

If the ASCII string is a variable, it can contain any number of characters up to the specified maximum. If the ASCII string is a data item, it must contain exactly the number of characters specified.

5.2.1.4 Numeric Types

Numeric data types represent numbers for computation. The numeric types are distinguished only by the length of the bit field used to contain the number. A BYTE is a 1-byte field, a WORD a 2-byte field, a LONGWORD a 4-byte field, and a QUADWORD an 8-byte field. For purposes of conversion, the numeric types are considered equivalent to strings, with the length determined by the type. (See Section 5.2.1.2.)

The special numeric type VALUE indicates a natural machine value. A VALUE is a WORD on a PDP-11 processor and a LONGWORD on a VAX processor.

Numeric types have a default output radix of decimal. The syntax for expressing a numeric type is as follows:

```
type [option[,...]]
```

The valid options are radix options and attribute options. The radix options determine the print radix. They are DECIMAL, OCTAL, HEX, BCD, BINARY, and RAD50. The attribute options are WIDTH and FILL.

The WIDTH option specifies the print field width and has the following format:

```
WIDTH=n
```

The FILL option specifies the fill character and has the following format:

```
FILL='character'
```

The default print field width and fill character for each choice of radix are as follows:

Radix	Fill	Print Field Width			
		BYTE	WORD	LONGWORD	QUADWORD
DECIMAL	space	3	5	10	20
OCTAL	'0'	3	6	11	22
HEX	'0'	2	4	8	16
BCD	'0'	2	4	8	16
BINARY	'0'	10	20	40	80
RAD50	space	N/A	3	6	12

The default radix is DECIMAL.

As an example, the specification for a LONGWORD to be printed in BCD using leading spaces and a field six characters wide is as follows:

```
LONGWORD BCD,WIDTH=6,FILL=' '
```

The special radix MACHINE is the normal radix used to express values for the host machine. The MACHINE radix is OCTAL for a PDP–11 processor and HEX for a VAX processor.

You can express numeric literals in a number of ways. A sequence of digits is, by default, interpreted as a VALUE numeric literal. The number is assumed to be decimal. You express a numeric literal of a specified type and radix as follows:

```
(# type_indicator 'character_string')
```

The character string is interpreted according to the specification given by the type indicator. The type indicator is a 1- or 2-character string specifying the type of the number and the radix in which to interpret the character string. The first character of the type indicator is the type of the number, as follows:

B Byte

W Word

L Longword

Q Quadword

V Value

The second character of the type indicator is the radix in which to interpret the character string. If the radix is not specified, the character string is assumed to be decimal. The valid radix indicators are the following:

A	ASCII
B	Binary
D	Decimal
I	Bit value
O	Octal
R	Radix–50
X	Hexadecimal

A minus sign (-) preceding any character string interpreted as a number indicates the two's complement of that number in the indicated radix; that is, binary, octal, decimal, and hexadecimal.

As an example, the following character string represents a byte that contains the octal value 17:

```
#BO'17'
```

The following character string represents a word containing the value -16_{10}:

```
#W'-16'
```

The bit-value radix indicator specifies that the quoted number is a decimal number representing a bit position. The value of the literal is 2 raised to the power of the bit position.

5.2.1.5 The FIELD Type

The FIELD type represents a field of numeric types. The BIT type represents a single bit of a numeric type and is equivalent to a LOGICAL type. The FIELD type represents a 1-or-more-bit field of a numeric type and is equivalent to a numeric type. A FIELD type is always a field of a numeric variable. There are no literal values for FIELD types.

The syntax for expressing a BIT type is as follows:

```
BIT[bit_number]
```

The syntax for expressing a FIELD type is as follows:

```
FIELD[low_bit_number:field_length in bits]
```

In either case, a FIELD type is declared directly following the numeric type of which it is a field.

5.2.1.6 The POINTER Type

The POINTER type is a table pointer. You use it to declare variables that temporarily store pointers for later use. The POINTER type cannot be converted to or from any other type. There is no literal representation of the POINTER type.

The value of the POINTER type is specific to a given table. A POINTER variable containing a value specifying a table entry for a given table cannot be used to reference an entry in another table. The variable can, however, be loaded with another value that references an entry in another table.

5.2.1.7 The RSX_TIME Type

The RSX_TIME type represents a time in RSX format. RSX time is represented as six sequential bytes containing the year since 1900, the month, day, hour, minute, and second in that order. This is a compression of the format returned by the Executive directive GTIM$.

The RSX_TIME type can only be printed or compared to other RSX_TIME types, or converted using one of the %TIM functions.

5.2.1.8 The VMS_TIME Type

The VMS_TIME type represents a time in VMS format. VMS time is represented as a quadword containing the time in hundreds of nanoseconds since 17 November 1858.

The VMS_TIME type can only be printed or compared to other VMS_TIME types, or converted using one of the %TIM functions.

5.2.2 Variables

The named variable is the fundamental unit for data manipulation. Named variables are defined in a given module, and available to that module and any modules called by the module. Named variables are declared in named groups. The full name of a variable is the name of the group, a period (.), and the name of the variable in the group, as follows:

```
group_name.variable_name
```

The group and variable names cannot be more than 15 characters in length. Names can include the letters A to Z, the numbers 0 to 9, the dollar sign ($), and the underscore (_). The leading character of a name must be alphabetic. You use the same syntax to reference data in either packets or subpackets.

The CFL compiler assigns each variable a type through declarations. (See Section 5.4 for a description of the declaration process.) Variables that are not fixed length, such as ASCII and STRING type variables, are assigned a maximum length as well. The variable can contain any amount of data that fits in its specified maximum length.

A field in the current record of a table can be referenced in the same manner as a variable, as follows:

```
table_name.field_name
```

The field value referenced is the specified field in the current record of the table. If there is no current record for the table, an error results.

Several special variables provide information about a group, packet, subpacket, or table. You reference the special variables in the following format:

```
group, [sub]packet or table_name.special_variable_name
```

The special variables are described as follows:

LENGTH

The length of the data in the group in addressable units of the host machine (bytes for PDP–11 and VAX processors). LENGTH includes all repetitions for repeated data or records for tables.

POINTER

Returns the current pointer for the specified group. POINTER is not valid for any data structure that would not have a current record context. This includes variables and non-repeated data. There is a current record context for tables and repeated data.

CONTEXT

Returns a logical value. If the specified group has a current record context, the value is set to true. If the specified group has no current record context, the value is set to false.

COUNT

Returns a numeric value representing the number of records in a group. For groups of variables, the value is always 1. For packets or subpackets, the value is the number of repetitions of the data. For tables, it is the number of records in the table.

5.2.3 Literals

Literal values can be assigned symbolic names. These symbolic names have the same syntax restrictions as variable names. Literal names are considered equivalent to their values in expressions. See Section 5.6.2 for information on the LITERAL statement.

5.2.4 Expressions

Expressions describe a computation through a sequence of operands and operators. Operands are variables or literals. Operators direct the computation. Expression evaluation is from left to right, and operator precedence is observed. Use open parentheses (( ) and close parentheses ( )) to override precedence.

Operators are either unary, which means that they take one operand, or binary, which means that they take two operands. Unary operators that precede the operand are called prefix operators; when they succeed the operand, they are called suffix operators. Binary operators are always specified between the two operands.

Operators are type-specific; that is, they operate between two elements of a specific type to produce a result of a specific type. The elements of an expression can be any of the following kinds:

literals

Literals express fixed values of a given type.

variables

Variables reference previously computed values.

(subexpressions)

Any valid expression enclosed in parentheses can be used as an element to an operator.

functions
A function is a predefined computation. (See Section 5.3.)

The following sections describe the various operators.

5.2.4.1 String Operators

String operators produce either binary or ASCII string results. The result is a string with the same type as the operand string or strings. The string operators are as follows:

- String concatenation (|) binary operator

  The string concatenation operator concatenates the first operand with the second. Both strings must be of the same type, as in the following expression:

  ```
  'ABC' | 'DEF'
  ```

  This expression produces the following string:

  ```
  'ABCDEF'
  ```

- Substring extraction (n:m) unary suffix operator

  The substring extraction operator produces the string formed by character n and the next m elements. String element numbers start with 1. For example, consider the following expression:

  ```
  'ABCDEFGH'<4:3>
  ```

  This expression produces the following string:

  ```
  'DEF'
  ```

  CFL treats both n and m as word values.

- Element extraction (n) unary suffix operator

  The element extraction operator is a special case of the substring extraction operator. It extracts the nth element as a single-character substring. For example, consider the following expression:

  ```
  'ABCDEFGH'<4>
  ```

  This expression produces the following string:

  ```
  'D'
  ```

5.2.4.2 Logical Operators

Logical operators perform operations on logical variables, or compare string or numeric variables to yield logical results.

The logical operators are as follows:

- Logical AND binary operator

  The logical AND operator does a logical AND of the two expressions. For example, the following expression produces the logical value false:

  ```
  TRUE AND FALSE
  ```

- Logical OR binary operator

  The logical OR operator does a logical OR of the two expressions. For example, the following expression produces the logical value true:

  ```
  TRUE OR FALSE
  ```

- Logical exclusive-OR (XOR) binary operator

  The logical exclusive-OR operator does a logical exclusive-OR of the two expressions. For example, the following expression produces the logical value false:

  ```
  TRUE XOR TRUE
  ```

- Logical NOT unary prefix operator

  The logical NOT operator produces the logical complement of a single variable. For example, the following expression produces the logical value false:

  ```
  NOT TRUE
  ```

- Bit extraction ([n]) unary suffix operator

  The bit extraction operator produces a logical result from numeric expressions. It is set to true if and only if bit n of the longword expression operand is set. For example, the following expression produces the logical value false:

  ```
  #WO'305' [4]
  ```

  The binary value of octal 305 is 011000110, with the fourth bit clear. Bits are numbered from 0.

5.2.4.3 Relational Operators

Relational operators compare string, time, or numeric operands. The comparisons are string comparisons if both operands are string or ASCII string operands. The comparisons are numeric comparisons if one operand is numeric and the other is either numeric or string. You cannot compare ASCII string operands and numeric operands. The comparisons are time comparisons if both operands are times of the same type. You cannot compare different types of time.

In numeric comparisons, the larger numeric value is greater.

In string comparisons, CFL stops at the first two characters that do not match and performs an ASCII sort. That is, CFL compares the ASCII values of the characters.

Note

Although Z is greater than A in ASCII, an ASCII sort is not the same as an alphabetic sort. Any lowercase letter has greater value than any uppercase letter, for instance, but any alphabetic character has greater value than any numeric character, and so forth.

If one string is longer than the other and the shorter string has the same leading elements as the longer, the longer string is greater.

In time comparisons, later times are greater.

The relational operators are as follows:

- Equality binary operator

  The equality (EQ) operator is set to true if and only if the operands are equal. For example, the following expression produces the logical value false:

  ```
  #WD'123' EQ #WD'355'
  ```

- Inequality binary operator

  The inequality (NE) operator is set to true if and only if the operands are not equal. For example, the following expression produces the logical value true:

  ```
  'ABCDEF' NE 'ABC'
  ```

- Greater-than binary operator

  The greater-than (GT) operator is set to true if and only if the first operand is greater than the second operand. For example, the following expression produces the logical value false:

  ```
  123 GT 355
  ```

- Less-than binary operator

  The less-than (LT) operator is set to true if and only if the first operand is less than the second operand. For example, the following expression produces the logical value true:

  ```
  'ABCDEF' LT 'ABCZZZ'
  ```

- Greater-than-or-equal binary operator

  The greater-than-or-equal (GE) operator is set to true if and only if the first operand is greater than or equal to the second operand. For example, the following expression produces the logical value true:

  ```
  45 GE 45
  ```

- Less-than-or-equal binary operator

  The less-than-or-equal (LE) operator is set to true if and only if the first operand is less than or equal to the second operand. For example, the following expression produces the logical value false:

  ```
  'Z' LE 'A'
  ```

- String-matching binary operator

  The string-matching (MATCH) operator compares strings. The strings are examined to determine which is shorter. The shorter string is compared character-by-character to the longer string. If all characters in the shorter string match with characters in the longer string, then the strings are equal and the value is set to true. This means a null string always matches any other string. For example, the following expression produces the logical value true:

  ```
  'ABCDEF' MATCH 'AB'
  ```

  The following expression produces the logical value false:

  ```
  'ABCDEF' MATCH 'ABCX'
  ```

5.2.4.4 Numeric Operators

Numeric operators operate on numeric variables as unsigned longwords. The numeric operators are as follows:

- Field extraction suffix unary operator

  The field extraction ([n:m]) operator produces the longword formed by taking the m-bit field beginning at bit n in the longword. Bit positions are numbered from least significant to most significant, beginning with 0. For example, the following expression produces the octal value 35:

  ```
  #WO'357' [3:6]
  ```

  Octal 357 has the binary value 011101111. Bit 3 and the next six bits have the binary value 011101, or octal 35.

- Logical SHIFT binary operator

  The logical SHIFT operator produces the first operand shifted by the number of bit positions specified by the second operand. Each left shift of one bit is the equivalent of multiplying by 2 and each right shift of one bit is the equivalent of dividing by 2. Indicate a left shift by making the second operand positive, and a right shift by making the second operand negative. If the second operand is zero, nothing shifts. The shifting is logical shifting; there is no sign extension on right shifts.

  For example, the following expression produces decimal 820, which is decimal 205 multiplied by 4:

  ```
  #WD'205' SHIFT 2
  ```

- Multiplication binary operator

  The multiplication (*) operator produces the product of the two operands. The result of the multiplication is truncated to the 32 low-order bits.

  For example, the following expression produces the decimal value 15:

  ```
  5 * 3
  ```

- Division binary operator

The division (/) operator produces the integer quotient of the two operands. For example, the following expression produces the decimal value 7:

```
15 / #B'2'
```

- Modulus binary operator

  The modulus (MOD) operator produces the remainder of the integer division of the two operands. For example, the following expression produces the decimal value 1:

  ```
  15 MOD 2
  ```

  A division or modulus operation with zero as the divisor causes an error.

- Addition binary operator

  The addition (+) operator produces the sum of the two operands. The sum is truncated to the 32 low-order bits. For example, the following expression produces the decimal value 17:

  ```
  5 + 12
  ```

- Subtraction binary operator

  The subtraction (-) operator, or minus sign, produces the difference of the two operands. The difference is truncated to the 32 low-order bits. For example, the following expression produces the decimal value 9:

  ```
  12 - 3
  ```

- Negation unary prefix operator

  The negation (-) operator, or minus sign, produces the two's complement of the operand. For example, the following expression produces the decimal value -8:

  ```
  - #B'8'
  ```

  The minus sign is both a unary and binary operator.

5.2.4.5 Bitwise Logical Operators

The bitwise numeric operators perform bitwise logical operations between two numeric operands. Rather than comparing the numeric operands as numbers, these operators compare the numeric operators bit by bit.

The bitwise logical operators are as follows:

- Bitwise AND binary operator

  The bitwise AND operator produces the bitwise logical AND of the two operands. For example, the following expression produces the octal value 1:

  ```
  #BO'41' AND #BO'3'
  ```

  The binary value of octal 41 is 00100001 and the binary value of octal 3 is 00000011. The bitwise AND operation determines that the least significant bit is set in both operands and returns the binary value 0000001, or octal 1.

- Bitwise OR binary operator

The bitwise OR operator produces the bitwise logical OR of the two operands. For example, the following expression produces the octal value 43:

```
#BO'41' OR #BO'3'
```

The bitwise OR of the binary values returns the binary value 00100011, or octal 43.

- Bitwise exclusive-OR (XOR) binary operator

  The bitwise XOR operator produces the bitwise logical exclusive-OR of the two operands. For example, the following expression produces the octal value 42:

  ```
  #BO'41' XOR #BO'3'
  ```

  The bitwise exclusive-OR of the binary values returns the binary value 00100010, or octal 42.

- Bitwise complement unary operator

  The bitwise complement (NOT) operator produces the bitwise complement (logical negation) of the operand. For example, the following expression produces the octal value 336:

  ```
  NOT #BO'41'
  ```

  The binary value of octal 41 is 00100001 and its complement is 11011110, or octal 3360.

5.2.5 Operator Precedence

Operations occur in the order defined by operator precedence unless overridden by parentheses. Operator precedence in CFL is the same as in most other languages (such as FORTRAN). Operators with higher precedence are evaluated before operators with lower precedence. For example, the following expression is evaluated as A + (B * C):

```
A + B * C
```

The expression is not evaluated as (A + B) * C because the multiplication operator (*) has higher precedence than the addition operator (+). In general, expressions are evaluated from left to right, taking into account operator precedence unless overridden by parentheses.

Operators are classified in the categories shown in the following table, and are listed in order of decreasing precedence. (The order in which operators are listed within a category is not significant.)

Highest Precedence Class	Prefix/Suffix Unary Operators
-	Numeric negation
NOT	Logical or numeric bitwise negation
[n:m]	Numeric field extraction
[n]	Logical value extraction
<n:m>	Substring extraction
<n>	Element extraction

Multiplication Precedence Class	Numeric Binary Operators
*	Numeric multiplication
/	Numeric division
SHIFT	Numeric logical shifting
MOD	Numeric modulus

Addition Precedence Class	Numeric Binary Operators
+	Numeric addition
-	Numeric subtraction

Logical Operation Class	Logical/Bitwise Logical Operators
AND	Logical and bitwise logical AND
OR	Logical and bitwise logical OR
XOR	Logical and bitwise logical XOR

Relational Class	Logical Comparison Operators
EQ	Equality
NE	Inequality
GT	Greater than
GE	Greater than or equal to
LT	Less than
LE	Less than or equal to

5.3 Functions

Functions provide special computations or special values not otherwise available to a program written in CFL.

A reference to a function in an expression has the following format:

```
%function [ (argument[=value][,...]) ]
```

Function names have the format %class name$function name. Some functions require arguments. *Note that the parentheses are a required part of the syntax when arguments are specified.* Functions return a value of a type that is fixed for a given function. For example, the following function returns the identification code of the current data packet as an ASCII string:

```
%PKT$IDENT
```

The following sections list the CFL functions by class name. The listings show the required syntax and are followed by descriptions of the values they return.

5.3.1 %CND Functions—Conditional Functions

The %CND functions select one of a set of expressions for evaluation. You can state criteria to select one of the arguments to be evaluated in a given context.

Note

All expressions are evaluated before determining a result. This means all expressions must be valid for any possible value of the logical expression. That is, %CND$IF is not entirely equivalent to an IF-THEN-ELSE statement, and %CND$SELECT is not entirely equivalent to a SELECT statement.

The %CND functions are as follows:

%CND$IF(logical, true exp, false exp)

This function evaluates the specified logical expression. If the expression is true, the true expression is returned as the value of the function. If the expression is false, the false expression is returned as the value of the function.

%CND$SELECT(selector, exp else, exp 0, [exp 1 ,...])

This function evaluates the specified selector expression. If the value of the expression is 0, the exp 0 expression is returned as the value of the function. If the value of the expression is 1, the exp 1 expression is returned as the value of the function. In general, if the value of the selector expression is n, the value of exp n is returned as the value of the expression. If no expression is provided corresponding to the value of the selector expression, the value of exp else is returned as the value of the expression.

5.3.2 %CNV Functions—Conversion Functions

The %CNV functions convert expressions to ASCII strings, primarily for printing. The conversions allow specification of the output radix, leading fill character (if any), and number of digits converted.

5.3.2.1 %CNV Functions—Numeric Conversion Functions

The numeric conversion functions convert numeric values to ASCII strings in the radix of the specific function. The syntax of these functions is as follows:

```
%CNV$xxx(numeric_value [,field_width [,fill_character]])
```

In this format, xxx is the radix. If no field width is specified, the default is 0. If no fill character is specified, the default is the null character. The field width and the fill character control the length of the returned string and justification of the digits in the string.

The numeric conversion functions behave as follows: CFL converts the numeric value to an ASCII string using the appropriate radix, and calculates the number of resulting digits. The following algorithm formats the returned string:

```
if field_width = 0 then
    return a string of length number_of_digits containing only the
        converted digits

else
    if number_of_digits > field_width then
        return a string of length field_width filled with asterisks

    else
        if fill_character = null_character then
            return a string of length field_width with the digits left
                justified and pad the string with trailing blanks

        else
            return a string of length field_width with the digits right
                justified preceded by the specified fill_character
```

The numeric conversion functions are as follows:

%CNV$OCTAL(numeric value[,field width[,fill character]])
Converts the number from binary to ASCII octal representation.

%CNV$DECIMAL(numeric value[,field width[,fill character]])
Converts the number from binary to ASCII decimal representation.

%CNV$DECIMAL_P(numeric value[,field width[,fill character]])
This function is identical to the %CNV$DECIMAL function, except that it appends a decimal point to the end of the output ASCII string. The decimal point is not counted in the field width.

%CNV$HEX(numeric value[,field width[,fill character]])
Converts the number from binary to ASCII hexadecimal representation.

%CNV$BCD(numeric value[,field width[,fill character]])
This function is identical to the %CNV$HEX function. It converts the number from binary to ASCII hexadecimal representation.

%CNV$BINARY(numeric value[,field width[,fill character]])
Converts the number from binary to ASCII binary representation.

%CNV$MACHINE(numeric value[,field width[,fill character]])

Converts the number from binary to ASCII representation in the natural machine radix, which is octal for a PDP–11 processor and hexadecimal for a VAX processor.

%CNV$RAD50(numeric value[,field width[,fill character]])

Converts a numeric type to an ASCII string using Radix–50 conversion. The numeric value must be a word, longword, or quadword.

5.3.2.2 %CNV Functions—Miscellaneous Conversion Functions

The following miscellaneous conversion functions are available:

%CNV$STRING(string)

Performs a hexadecimal conversion of the specified string to ASCII.

%CNV$RSX_TIME(RSX_time_value[,fields])

Converts the RSX time value to a string in the format yy-mmm-dd hh:mm:ss. The optional fields parameter specifies the number of fields of the date to be converted. To convert only the date, specify 3. To convert the date and time (excluding the seconds), specify 5. The default is the full date and time expressed in six fields.

%CNV$VMS_TIME(VMS_time_value[,fields])

Converts the VMS time value to a string in the format yy-mmm-dd hh:mm:ss. The optional fields parameter specifies the number of fields of the date to be converted. To convert only the date, specify 3. To convert the date and time (excluding the seconds), specify 5. The default is the full date and time expressed in six fields.

5.3.3 %COD Functions—Encoding Functions

The encoding functions convert ASCII strings into numeric values, using different radixes. The encoding functions are as follows:

%COD$OCTAL(string)

Converts the string to a value using octal radix. The string may contain only the digits 0 to 7 and optional leading spaces or the minus sign (-).

%COD$DECIMAL(string)

Converts the string to a value using decimal radix. The string may contain only the digits 0 to 9 and optional leading spaces or the minus sign (-).

%COD$HEX(string)

Converts the string to a value using hexadecimal radix. The string may contain only the digits 0 to 9, the letters A through E, and optional leading spaces or the minus sign (-).

%COD$BCD(string)

Same as %COD$HEX. Converts the string to a value using hexadecimal radix. The string may contain only the digits 0 to 9, the letters A through E, and optional leading spaces or the minus sign (-).

%COD$BINARY(string)

Converts the string to a value using binary radix. The string may contain only the digits 0 and 1, and optional leading spaces or the minus sign (-).

%COD$MACHINE(string)

Converts the string to a value using the natural radix for MACHINE, which is octal for a PDP–11 processor and HEX for a VAX processor.

%COD$RSX_TIME(string)

Converts the string to a date in RSX format. The string must be of the form dd-mmm-yy [hh:mm[:ss]]. The date and time can occur in either order; the seconds (:ss) are optional. The default for the time fields is 00:00:00.

%COD$VMS_TIME(string)

Converts the string to a date in VMS format. The string must be of the form dd-mmm-yy hh:mm:ss. The date and time can occur in either order; the seconds (:ss) are optional.

5.3.4 %COM Functions—Computational Functions

The computational functions are as follows:

%COM$AND(numeric expression, numeric expression)

Returns the logical AND of the two numeric expressions. Both expressions must be machine values or shorter. This function is used primarily for overlay reasons on the PDP–11 processor.

%COM$HARDWARE(numeric expression)

Returns the ASCII character corresponding to the numeric expression in the DIGITAL hardware alphabet, which is ABCDEFHJKLMNPRSTUV, numbered from 0 to 17. For example, %COM$HARDWARE(0) returns an A.

%COM$LONGWORD(value,bit [,value,bit [,...]])

Returns a longword value. Each value is shifted by the specified number of bits and then the logical OR of all the values is returned.

%COM$NEGATE(value)

Returns the negative of the specified value. This is the two's complement of the value.

%COM$NULL(expression)

Returns a true value if the result of the expression is a value of length zero, or a false value if the result of the expression is a value with length other than zero.

5.3.5 %CTL Functions—RPT Control Functions

The RPT control functions are as follows:

%CTL$OPEN(file,filespec,default_spec)

Opens the file using the file specification and the default file specification. The value of the function is the fully qualified file specification for the file.

%CTL$STATUS(file)

Returns the value set to true if the file is open and set to false if the file is not open.

%CTL$FILE_STATUS

Returns the numeric status value returned by the file system after the last file-open operation.

%CTL$EOF(file)

Returns the value set to true if the specified file is at end-of-file (EOF), or set to false if the file is not at EOF.

%CTL$CLOSE(file)

Closes the file. The value of the function is the number of records written to the file.

%CTL$INPUT(low,high)

Sets the lowest and highest packets to be processed by RPT. Returns a true value if both packet specifications are syntactically correct, and a false value if neither is syntactically correct.

This implicitly sets the processing direction as well because, if the high packet is lower than the low packet, the file is processed backwards. A null packet specification takes the default: the beginning of the file for the low packet and the end of the file for the high packet.

5.3.6 %LOK Functions—Look-Ahead Functions

The %LOK functions obtain information in undeclared data packets or subpackets. There is a %LOK function for each of the data types supported for look-ahead operations. All offsets are byte offsets. The %LOK functions are as follows:

%LOK$BYTE(subpacket_number,offset)

Returns the specified byte from the current data packet. The subpacket number is the number of the subpacket from which the data is to be obtained. If the subpacket number is 0, the data is obtained from the packet itself. The offset is the byte offset in the subpacket for the data item.

%LOK$WORD(subpacket_number,offset)

Returns the specified word from the current data packet. The subpacket number is the number of the subpacket from which the data is to be obtained. If the subpacket number is 0, the data is obtained from the packet itself. The offset is the byte offset in the subpacket for the data item.

%LOK$LONGWORD(subpacket_number,offset)

Returns the specified longword from the current data packet. The subpacket number is the number of the subpacket from which the data is to be obtained. If the subpacket number is 0, the data is obtained from the packet itself. The offset is the byte offset in the subpacket for the data item.

%LOK$LENGTH(subpacket_number)

Returns the length of the data in the specified subpacket. The subpacket number is the number of the subpacket whose length is to be returned. If the subpacket number is 0, the length of the data packet is returned.

5.3.7 %PKT Functions—Packet Information Functions

The following %PKT functions obtain information about the current packet:

%PKT$MODULE(module_name)

Returns the value true if the specified module exists in the control file, and false if it does not exist.

%PKT$IDENT

Attempts to get the next packet from the input file in the range specified by %CTL$INPUT and makes it the current packet. If no more packets exist within that range, a null string is returned. Otherwise, %PKT$IDENT returns the current packet identification as a fixed-length ASCII string of eight characters.

5.3.8 %RPT Functions—Report Control Functions

The following %RPT functions control report generation:

%RPT$PAGE_SIZE(lines)

The default page size is 57_{10} lines of text plus headers and a form feed. %RPT$PAGE_SIZE changes the number of lines per page to the specified value. If the value is 0, the page size is infinite. The function returns the previous number of lines per page before the function was executed.

%RPT$PAGE_DEFAULT

Returns the default number of lines per page of RPT, which is 57_{10}.

%RPT$PAGE_CURRENT

Returns the current number of lines per page.

%RPT$PAGE_REMAINING

Returns the number of lines remaining on the current page.

%RPT$LINE_SKIP(interval,lines)

Causes RPT to skip the specified number of lines for every interval number of lines. If the interval is 0, automatic line skipping is suppressed. The function value is the previous interval.

%RPT$LINE_REMAINING

Returns the number of lines remaining in the current interval.

%RPT$COMMAND

Returns the command line as a string.

%RPT$IDENT

Returns the RPT identification as a string.

%RPT$STATUS(status)

Sets the exit status of RPT to the specified status value if it is more severe than the current exit status. If it is not more severe, no action is taken. The actual status value is determined by the control files using this function, based on the value given by the status argument. A status value is considered a SUCCESS, or true, status if the low bit is 1, and a FAILURE, or false, status if the low bit is 0.

The following algorithm, where NEW_STATUS is the value of the status argument and EXIT_STATUS is the current exit status, is used to update the exit status:

```
IF  NEW_STATUS
THEN
    BEGIN
    IF EXIT_STATUS AND ( NEW_STATUS GT EXIT_STATUS )
    THEN
         SET EXIT_STATUS TO NEW_STATUS ;
     END ;
ELSE
    BEGIN
    IF EXIT_STATUS OR ( NEW_STATUS GT EXIT_STATUS )
    THEN
         SET EXIT_STATUS TO NEW_STATUS
    END ;
```

The function returns the original value of EXIT_STATUS rather than the potentially updated EXIT_STATUS.

5.3.9 %STR Functions—String-Handling Functions

The following %STR functions manipulate ASCII and binary strings:

%STR$TRAIL(string,element)

Removes all trailing repetitions of the specified element from the specified string. The value of the function is the original string without the trailing characters.

%STR$LENGTH(string)

Returns the length of the specified string as a numeric value.

%STR$PARSE(string,pointer,control)

Performs a simple parse by returning a pointer to the end of the substring, beginning at the specified pointer position in the string and terminated by any of the characters in the control string or by the end of the string.

%STR$QUOTE(string,pointer,control,quote)

Performs a simple parse with quote characters. STR$QUOTE works the same way as STR$PARSE except that STR$QUOTE handles quote characters. The quote argument is a character string of two characters. For clarity, the two characters should match in some way, but this is not required. If the first character of the pair is encountered, checking for control characters stops until the second character of the pair appears. For example, the quote string ' <> ' causes anything between a left angle bracket (<) and a right angle bracket (>) to be considered as "quoted" and treated as a unit.

%STR$REMAINING(string,pointer)

Returns the substring of the specified string consisting of all characters including and following the specified pointer position.

%STR$MATCH(string,string)

Performs an element-by-element comparison of the two strings. The comparison continues only as long as there are elements to compare. That is, with strings of different lengths, the comparison stops with the last element in the shorter string. %STR$MATCH returns a value set to true if the elements match, and false if they do not.

%STR$SEARCH(string,pointer,string)

Searches the first string, beginning at the specified pointer position, for the second string. The pointer returns to the position in the first string at which the second string begins. If the second string is not found, %STR$SEARCH returns a zero.

%STR$PAD(string,paddingstring,lead,trail)

Creates a new string consisting of the specified string padded with the single-character padding string. The lead and trail numeric expressions specify how many padding characters you want to lead or trail the original string.

%STR$FILE(string,pointer)

Assumes the pointer is at the beginning of the file specification. It returns a pointer to the character following the file specification. If the string pointed to is not a valid file specification, %STR$FILE returns a zero.

%STR$UPCASE(value)

Returns the specified string with all lowercase letters converted to uppercase.

%STR$CHAR(value)

Returns the character corresponding to the specified value.

5.3.10 %TIM Functions—Time-Handling Functions

The %TIM functions manipulate times. The time values include both date and time unless otherwise noted. The RSX and VMS %TIM functions are as follows:

RSX Time Functions

%TIM$RSX_CURRENT

Returns the current date and time as a value in RSX format.

%TIM$RSX_DATE(RSX_time_value)

Returns the date only in RSX format.

%TIM$RSX_VMS(VMS_time)

Returns an RSX time value corresponding to the specified VMS time.

%TIM$RSX_NULL

Returns a null RSX time value. This value prints as all blank spaces.

VMS Time Functions

%TIM$VMS_CURRENT
Returns the current date and time as a value in VMS format.

%TIM$VMS_DATE(VMS_time_value)
Returns the date only in VMS format.

%TIM$VMS_PLUS(VMS_time,days)
Returns a VMS time value containing the specified time plus the specified number of days.

%TIM$VMS_MINUS(VMS_time,days)
Returns a VMS time value containing the specified time minus the specified number of days.

%TIM$VMS_RSX(RSX_time)
Returns a VMS time value corresponding to the specified RSX time.

%TIM$VMS_NULL
Returns a null VMS time value. This value prints as all blank spaces.

5.3.11 The %USR Function—User I/O Function

The %USR function performs input and output to and from the user of RPT, as follows:

%USR$STRING(file,prompt,maximum_length)
Writes out the prompt string if the specified file is a terminal, and reads a string input whose maximum length is specified by the length parameter. If the specified output file is not a terminal, there is no prompt and only the read is performed.

5.4 Declarations

This section describes the declaration of variables and data items. A declaration includes print-formatting information along with the definition of data items; it is different from declarations in most languages.

5.4.1 Scope of Declarations

Data items can be referred to during the scope of the declaration; that is, from the point they are declared until the declaration is discarded. If a declaration is made in a given procedure, data items can be referenced in the defining procedure or any procedure called by it.

5.4.2 The DECLARE Statement

The DECLARE statement begins the declaration of a block of variables. The format of the DECLARE statement is as follows:

```
DECLARE group name [ NAMED ] ;
       variable_name : type [: print_expressions] ;
       variable_name : type [: print_expressions] ;
       variable_name : type [: print_expressions] ;
END_DECLARE ;
```

group name

The group name is the prefix name by which the DECLARE statement identifies the group variables. The name of each group variable is in the following format:

```
group_name.variable_name
```

type

The variable type is one of the RPT data types: LOGICAL, STRING, ASCII, NUMERIC, FIELD, RSX_TIME, or VMS_TIME.

NAMED

The optional NAMED qualifier specifies that the symbol names are to be kept and used with the WRITE_GROUP statement FORMAT clause qualifiers that print a symbol name.

print_expressions

The optional print expressions specify expressions to be evaluated and printed if the group is printed using the WRITE_GROUP statement. If you specify more than one expression, separate them by commas.

The print information consists of one or more expressions separated by commas. If the WRITE_GROUP statement is used to print the data group, the print expressions are evaluated and printed.

Declaration of a numeric type can be followed by declaration of one or more field types. The field types are considered fields of the preceding numeric type declaration.

Print information for field types is handled specially. The variable names for field types do not appear when the WRITE_GROUP statement is used, and the print expressions following a field type declaration are considered to apply to the preceding numeric type. Of the print expressions following a field type, one is selected based on the field value.

For BIT fields, the first print expression is used if the BIT is set to true, and the second if the BIT is set to false.

For FIELD fields, the first print expression is used if no other print expression applies. The second print expression is used if the FIELD value is 0, the third if the FIELD value is 1, the fourth if the FIELD value is 2, and so on.

When a print expression is printed for a field type, it is printed in the following format:

```
p[h:l] (t[...])
```

Parameters

p — The leading character of the print expression. This appears as a prefix to the print field.

h — The high-bit number of the FIELD, or the bit number of the BIT field. The square brackets are printed.

l — The low-bit number of the FIELD. The "l" field and the leading colon are printed as blanks for BIT fields.

t The trailing characters of the print expression. The trailing characters are any characters following the first character.

The following example illustrates a print expression:

```
DECLARE EXAMPLE :
VARIABLE_1 : WORD :
: FIELD [6:2] : '*The value of this field is 2 or 3',
' The value of this field is 0' ,
' The value of this field is 1' ;
```

When EXAMPLE is printed with a WRITE_GROUP statement, the field will be printed as follows, depending on whether bits 6 and 7 contain the value 0, 1, 2, or 3:

```
[ 7: 6]      The value of this field is 0

[ 7: 6]      The value of this field is 1

*[ 7: 6]     The value of this field is 2 or 3
```

Note

The field is declared in the form [6:2], meaning that it starts at bit 6 and is 2 bits long. However, the print format is expressed in the form [7: 6], meaning that it consists of bits 6 and 7.

5.4.3 The PACKET Statement

The PACKET statement declares an input data packet. The format is as follows:

```
PACKET name [ REPEATED ] [ NAMED ] ;
       name : type : print_information ;
       name : type : print_information ;
       name : type : print_information ;
END_PACKET ;
```

The REPEATED data attribute is optional. A PACKET statement without this attribute specifies a single packet. The REPEATED attribute specifies that the data in the packet is repeated. The number of repetitions is computed by dividing the packet length by the length of the data items. Note that the items must be referenced as for a DYNAMIC_TABLE statement; they cannot be referenced directly.

The optional NAMED qualifier specifies that the symbol names are to be kept and used with the WRITE_GROUP statement FORMAT clause qualifiers that print a symbol name.

The declaration of data defines the special variable LENGTH, referenced as the data items themselves would be referenced, as follows:

```
data_group_name.LENGTH
```

Note that the length can be referenced directly, even for a REPEATED data group.

Each of the data item names is declared, along with the type and print information. The name is the name by which the data element is referenced.

The format for a data element reference is as follows:

```
data_group_name.data_element_name
```

The data item type is declared as specified in Section 5.2.1.

The special variable name RESERVED in place of an element declaration specifies a sequence of undefined values. The type declaration specifies the length of the undefined area. This cannot be a field type. The syntax is as follows:

```
PACKET name data organization ;
       name : type : print_information ;
       RESERVED : type ;
       name : type : print_information ;
END_PACKET ;
```

A RESERVED declaration in a PACKET or SUBPACKET statement indicates to the compiler (and RPT) that the area is currently unused, but to use its length in determining the size of the packet or subpacket and the offsets of elements following the RESERVED declaration. You use RESERVED either to reserve space for future use or to force word-boundary alignments.

Note that an element name could be used in these situations, but that RESERVED serves as a documentation aid and saves having to define unique element names if there are multiple unused areas in a packet or subpacket.

5.4.4 The SUBPACKET Statement

The SUBPACKET statement declares an input data subpacket. The format is as follows:

```
SUBPACKET name = expression  <data attribute> [ NAMED ] ;
         name : type [: print_information] ;
         name : type [: print_information] ;
         name : type [: print_information] ;
END_PACKET ;
```

The attribute, if present, is REPEATED. A SUBPACKET statement without an attribute specifies a single subpacket. The leading REPEATED attribute specifies that the data in the subpacket is repeated. The number of repetitions is computed by dividing the subpacket length by the length of the data items. Note that the items must be referenced as for a DYNAMIC_TABLE statement; they cannot be referenced directly.

The optional NAMED qualifier specifies that the symbol names are kept and used with the WRITE_GROUP statement FORMAT clause qualifiers that print a symbol name.

The handling of a SUBPACKET statement is otherwise the same as for a PACKET statement.

5.4.5 Conditional Declarations

RPT provides a mechanism for the conditional declaration of data items. Conditional declaration can only be used for data; that is, PACKET and SUBPACKET declarations. FIELD and BIT declarations cannot cross conditionals. All FIELD and BIT declarations must be in the same conditional as their data item.

The conditionals allowed in declarations are as follows:

- IF has the following syntax:

```
name : type [: print_information] ;
name : type [: print_information] ;
name : type [: print_information] ;

IF expression
THEN
       BEGIN
       name : type [: print_information] ;
       name : type [: print_information] ;
       name : type [: print_information] ;
       END
ELSE
       BEGIN
       name : type [: print_information] ;
       name : type [: print_information] ;
       name : type [: print_information] ;
       END
END_IF ;

name : type [: print_information] ;
name : type [: print_information] ;
name : type [: print_information] ;
```

If the expression is set to true, the THEN clause is defined. If the expression is set to false, the ELSE clause is defined. The ELSE clause is optional.

- CASE has the following syntax:

```
name : type [: print_information] ;
name : type [: print_information] ;
name : type [: print_information] ;

CASE expression OF
       [expression [,...]] :
              BEGIN
              name : type [: print_information] ;
              name : type [: print_information] ;
              name : type [: print_information] ;
              END
       [ expression [,...]] :
              BEGIN
              name : type [: print_information] ;
              name : type [: print_information] ;
              name : type [: print_information] ;
              END
       [expression , expression , ... ] :
              BEGIN
              name : type [: print_information] ;
              name : type [: print_information] ;
              name : type [: print_information] ;
              END

       ELSE
```

```
        BEGIN
        name : type [: print_information] ;
        name : type [: print_information] ;
        name : type [: print_information] ;
        END

    END_CASE ;

    name : type [: print_information] ;
    name : type [: print_information] ;
    name : type [: print_information] ;
```

The expression in the CASE statement is evaluated and the expression lists searched to find a matching expression value. If a match is found, the declaration is made. If no matching expression value is found, the optional ELSE clause is executed. Otherwise, an error occurs.

5.5 Action Statements

Action statements perform processing. CFL has a limited set of action statements because it is a simple, special-purpose language. The statements provided have capabilities designed to make the handling of error log data as simple as possible.

5.5.1 The SET Statement

SET sets the value of a variable to the results of a computation. The format is as follows:

```
SET variable_name TO expression ;
```

The expression is evaluated using the type of the specified variable, and the variable is set to the value of the expression.

5.5.2 The INCREMENT and DECREMENT Statements

INCREMENT and DECREMENT adjust the value of numeric variables of length VALUE (a word on a PDP-11 processor, a longword on a VAX processor) or less by a value. The value defaults to 1. The format of the statements is as follows:

```
INCREMENT variable_name [ BY numeric_expression ] ;

DECREMENT variable_name [ BY numeric_expression ] ;
```

The value of the variable is increased by the value of the numeric expression for the INCREMENT statement, and decreased by the value of the numeric expression for the DECREMENT statement.

5.5.3 The WRITE Statement

WRITE writes information to a specified output location. The format is as follows:

```
WRITE (expression [,...]) TO output FORMAT format ;
```

The expressions are printed in the order specified. The optional output clause can contain either the specifications REPORT or ERROR. ERROR directs output to the invoking terminal. REPORT is the default and directs output to the output report. The REPORT file must be open. The ERROR file is always open. The format for printing the output report is described in Section 5.10.

5.5.4 The WRITE_GROUP Statement

WRITE_GROUP writes a decoded block of data. The data definitions define the formatting. The format is as follows:

```
WRITE_GROUP group_name TO output FORMAT format, format ;
```

The group name is the name of the group of variables or data items to be written. The optional output specification is the same as for the WRITE statement and has the same defaults. The format for printing the output data is described in Section 5.10.

The group name can be followed by a symbol list, as follows:

```
group_name (symbol_name[,...])
```

In this case, only the specified symbols are listed.

The first FORMAT clause is for printing all data items, the second FORMAT clause is for printing all BIT and FIELD items.

5.5.5 The DECODE Statement

The DECODE statement performs specialized declaration-to-text translation. The statement has the following form:

```
DECODE       variable_name = group name ;
       <data-item> [ <bit-number> ] ;
       <data-item> [ <bit-number> ] ;
       <data-item> [ <bit-number> ] ;
END_DECODE ;
```

Each of the data items must be a data item in the specified group. The bit numbers are numbers of bits in the data item. The DECODE statement processes the data items in the order specified, checking the specified bits. If a bit is found to be set to true, the corresponding bit-to-text translation for that bit is performed, and the result returned in the specified variable. This completes the statement. A data item can be preceded by a NOT, which indicates the specified bit must be set to false for the bit-to-text translation to be performed. If no bit-to-text translation is found, the null string is returned.

5.6 Control Statements

Control statements direct RPT by defining and invoking procedures and controlling termination of the procedure. Other control statements also conditionally control the execution of statements in a procedure. These are called conditional statements.

5.6.1 The MODULE Statement

MODULE declares the name of the module being compiled. The format is as follows:

```
MODULE name ident ([attribute [,...]]) ;
```

Parameters

MODULE

MODULE must be the first statement in any module. Each module must end with an END_MODULE ; statement.

name

The module name specifies the name to be used when the module is inserted into the control file library.

ident

The module ident is a quoted string to be inserted in the module header.

attribute

The optional module attributes specify information to be used in processing the module. The following two attributes are recognized:

KEEP

Specifies that if a module cache is used by RPT, the module should be kept because it is likely to be used again.

FLUSH

Specifies that if a module cache is used by RPT, the module should be flushed because it is unlikely to be used again.

5.6.2 The LITERAL Statement

The LITERAL statement assigns a name to a literal value. All LITERAL statements in a module must precede any PROCEDURE statement. The format of a LITERAL statement is as follows:

```
LITERAL group.name = compiletime_constant_expression ;
```

The name is equivalent to the value of the compiletime constant expression, and can be used in any expression or compiletime constant expression to represent the specified value.

5.6.3 The CALL Statement

CALL invokes a subroutine. CALL has the following format:

```
CALL [MODULE module_name_expression]
       PROCEDURE procedure_name_expression
       [COROUTINE procedure_name_expression] ;
```

Parameters

MODULE

The optional module name expression specifies the module to be called. If the module name is not specified, the specified procedure is assumed to be in the current module.

PROCEDURE

The procedure name specifies the procedure to call.

COROUTINE

The optional COROUTINE argument specifies that the two called procedures are coroutines. The first called procedure is specified as in the normal form of a CALL statement. The procedure specified using the COROUTINE argument is executed first. That procedure can then execute a statement that passes control to the other procedure specified in the CALL statement. None of the declarations is lost.

The two procedures can trade control back and forth using the COROUTINE statement. Each time a COROUTINE statement is executed, the other procedure resumes execution from the point of the last COROUTINE statement. If one of the procedures returns, control passes to the other procedure as if a COROUTINE statement were executed. When both procedures have completed, the coroutines exit to the caller.

5.6.4 The RETURN Statement

The RETURN statement forces a return from a procedure to the calling procedure. RETURN is optional at the end of a procedure. The format of the RETURN statement is as follows:

```
RETURN ;
```

The RETURN statement terminates the current procedure and returns control to the calling procedure.

5.6.5 The PROCEDURE Statement

The PROCEDURE statement declares the beginning of a procedure. It has the following format:

```
PROCEDURE procedure_name statement_block ;
```

Parameters

procedure name

The procedure name cannot be more than 15 characters in length. Names can include the letters A to Z, the numbers 0 to 9, the dollar sign ($), and the underscore (_). The leading character of a name must be alphabetic.

statement block

The statement block is executed as the named procedure.

5.6.6 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement is the most basic conditional statement. (Other conditional statements are provided to simplify the handling of common situations that would be cumbersome with IF-THEN-ELSE.) The IF-THEN-ELSE statement has the following format:

```
IF logical_expression THEN block ;
[ELSE block;] END_IF ;
```

If the logical expression is set to true, the block following the THEN statement is executed. If the logical expression is set to false, the block following the ELSE statement is executed. The ELSE clause is optional. If it is not specified, no action is performed if the expression is set to false.

Each block consists of a single statement. Using BEGIN-END statements, a block can contain a compound statement. See Section 5.6.11 for a description of how to make multiple statements appear as a single logical entity to conditional statements by using BEGIN-END statements.

5.6.7 The CASE Statement

The CASE statement selects one of a set of possible outcomes based on an expression. The format of a CASE statement is as follows:

```
CASE expression OF
   [ expression ,... ] : block ;
   [ expression ,... ] : block ;
   [ expression ,... ] : block ;
[ELSE block ;]
END_CASE ;
```

CASE executes the block corresponding to the first expression equal to the numeric expression. If ELSE is specified and no expression matches, it is executed.

5.6.8 The SELECT Statement

The SELECT statement is a variation of CASE. SELECT selects one of a given set of blocks. The general format is as follows:

```
SELECT numeric_expression OF
    block ;
    block ;
    block ;
[ELSE block ;]
END_SELECT;
```

SELECT selects the nth block, where n is the value of the numeric expression and is greater than or equal to 1. If the last block is preceded by ELSE, the block is executed if and only if the value of the numeric expression exceeds the number of blocks supplied.

5.6.9 The WHILE, UNTIL, and DO Statements

The WHILE, UNTIL, and DO statements control conditional looping. To specify a conditional loop, specify a block of statements to be conditionally executed and an expression to control the execution. The following conditions apply to these statements:

- The DO statement specifies the block of statements to be conditionally executed.
- The WHILE statement specifies an expression to be considered satisfied if it is set to true.
- The UNTIL statement specifies an expression to be considered satisfied if it is set to false.
- A DO statement must be specified with a WHILE or UNTIL statement. The block of statements specified by the DO statement is executed until the condition specified by the WHILE or UNTIL statement is no longer satisfied.
- If a DO WHILE or DO UNTIL statement is specified, the DO statement is executed once before testing the condition. If a WHILE DO or UNTIL DO statement is specified, the condition is tested first before executing the DO statement.

You can use the WHILE, UNTIL, and DO statements in the following combinations:

```
DO block WHILE expression ;
```

Executes the block once, then evaluates the expression. If the expression is set to true, the block is repeated. If the expression is set to false, execution continues following the WHILE statement.

```
DO block UNTIL expression ;
```

Executes the block once, then evaluates the expression. If the expression is set to false, the block is repeated. If the expression is set to true, execution continues following the UNTIL statement.

```
WHILE expression DO block
```

Evaluates the expression. If the expression is set to true, the block is executed and the process repeated. If the expression is set to false, execution continues following the DO statement.

```
UNTIL expression DO block
```

Evaluates the expression. If the expression is set to false, the block is executed and the process repeated. If the expression is set to true, execution continues following the DO statement.

5.6.10 The LEAVE Statement

The LEAVE statement immediately terminates the current DO statement. The control expression in the associated UNTIL or WHILE statement is considered satisfied and is not reevaluated.

5.6.11 The BEGIN-END Statement

The BEGIN-END statement forces a compound statement to be treated as one statement for purposes of conditionals. For example, to process two statements in the THEN clause of an IF statement, use the the following construct:

```
IF logical expression
THEN
       BEGIN
              statement 1 ;
              statement 2 ;
       END ;
ELSE
       statement ;
END_IF ;
```

5.6.12 Lexical Conditionals

Lexical conditionals perform conditional handling at compilation. A lexical conditional is valid wherever a statement is valid. Lexical conditionals have the following format:

```
$IF compiletime_constant_expression
$THEN
       statement_block
[$ELSE
       statement_block]
$END_IF
```

The $ELSE block is optional. If the compiletime constant expression is set to true, everything in the $THEN block is compiled and the $ELSE block is not compiled. If the compiletime constant expression is set to false, the $THEN block is not compiled and the $ELSE block, if present, is compiled.

Lexical conditionals can be nested to any level.

5.7 Tables

The table is one of the fundamental units of data organization for RPT. Tables are used to structure large amounts of data to be referenced during report generation.

5.7.1 Table Structure

Tables are sets of similar records containing fields by which the records can be referenced. Tables and the data in them can either be declared statically as part of the definition of a given control file module, or they can be declared dynamically during the operation of RPT. Static tables hold reference data, while dynamic tables store information computed during the operation of RPT.

Each record in a table is a sequence of named fields. The definition of the table defines the names of the fields and their sequence.

You refer to tables by name. Table names follow the ordinary rules for naming groups. The name cannot be more than 15 characters in length. Names can include the letters A to Z, the numbers 0 to 9, the dollar sign ($), and the underscore (_). The leading character of a name must be alphabetic.

Fields in a table are also named. Field names follow the same rules.

Table entries are manipulated by setting the current entry pointer for a table, and then using either the table manipulation statements or simple variable references to read or modify the data in the table.

The following sections describe each of the table-definition and table-manipulation statements in detail.

5.7.2 The TABLE Statement

The TABLE statement defines a static table. The format of the statement is as follows:

```
TABLE table name ;
       name : type [: print_expressions] ;
       name : type [: print_expressions] ;
       name : type [: print_expressions] ;
BEGIN_TABLE
       value, value, [,...] ;
       value, value, [,...] ;
       value, value, [,...] ;
END_TABLE ;
```

The declaration list following the TABLE statement specifies each of the fields, their types, and print information. The format is the same as for the DECLARE statement. The list values are individual compile-time constant expressions separated by commas. Each sequence of list values separated by commas (,) and terminated by a semicolon (;) represents one TABLE record. Each table value must be of the same type as the corresponding declaration from the declaration list. TABLE records cannot be modified at run time.

5.7.3 The DYNAMIC_TABLE Statement

The DYNAMIC_TABLE statement declares a dynamic table. The format of a DYNAMIC_TABLE statement is as follows:

```
DYNAMIC_TABLE table name ;
       name : type [: print_expressions] ;
       name : type [: print_expressions] ;
       name : type [: print_expressions] ;
END_TABLE ;
```

Records are placed in the dynamic table at run time through the use of the PUT statement, and can be modified by some of the POINTER statements. (See Sections 5.7.5 and 5.7.7.)

5.7.4 The FILE Statement

The FILE statement is identical to the DYNAMIC_TABLE statement. It is included for compatibility only. You should use DYNAMIC_TABLE because the FILE statement may be removed in a future release.

FILE declares a dynamic table. The format of a FILE statement is as follows:

```
FILE table name ;
        name : type [: print_expressions] ;
        name : type [: print_expressions] ;
        name : type [: print_expressions] ;
ENDFILE
```

Records are placed in the file dynamically at run time through use of the PUT statement and can be modified by some of the POINTER statements.

5.7.5 The POINTER Statement

The POINTER statement adjusts the current pointer for a table. It uses the following syntax:

```
POINTER table_name action [pointer_variable] ;
```

The argument action can be one of the following:

FIRST
Sets the current table pointer to the first record of the table. If there is no next record, then the current table pointer is set to null (see RESET).

NEXT
Sets the current table pointer to the next record of the table. If there is no next record, then the current table pointer is set to null (see RESET).

PREVIOUS
Sets the current table pointer to the previous record of the table. If you back up past the beginning, then the table pointer is set to null (see RESET).

RESET
Sets the current table pointer to null; that is, there is no table pointer.

LOAD pointer variable
Sets the current table pointer to the value of the pointer variable.

CLEAR

The specified table must be a dynamic table (see Section 5.7.3). Deletes all records from the dynamic table and sets the current table pointer to null (see RESET).

DELETE

The specified table must be a dynamic table. Deletes the current record and advances the pointer to the next record. If there is no next record, then the current table pointer is set to null (see RESET).

MOVE pointer variable

The specified table must be a file (see Section 5.7.4). The record pointed to by the pointer variable is moved to the current record position, and the current record and all following records are moved up one record. This is used mainly for sorting records in a file.

5.7.6 The FIND Statement

The FIND statement finds a record in a table using one or more key values. The format of the FIND statement is as follows:

```
FIND table_name field=value[,...]SELECT expression ;
```

The table is searched until an entry with all specified fields having the specified value is encountered. Tables are searched sequentially from the current pointer position. If no record is found, the current pointer for the table is set to null.

If you specify the optional SELECT clause, a record does not satisfy the search criteria unless the select expression, evaluated with the current record for the table set to the specified record, is set to true.

5.7.7 The PUT Statement

The PUT statement creates a new record in a table. The specified table must be a dynamic table (see Section 5.7.3). The PUT statement has the following format:

```
PUT table_name field=expression , ... ;
```

Sets the specified fields of the record to the values of the specified expressions. Note that all fields must be specified; none of the fields of the record is optional.

5.8 Lists

This section describes how CFL handles expression lists.

5.8.1 The LIST Statement

The LIST statement declares a list of expression groups. The format of the LIST statement is as follows:

```
LIST list_name ;

expression [,...] ;
expression [,...] ;
expression [,...] ;

END_LIST ;
```

The expression lists can then be referenced by the SIGNAL, SIGNAL_STOP, and MESSAGE statements, which are described in Section 5.9.

5.8.2 The SEARCH Statement

SEARCH locates a specific entry in a list. The SEARCH statement has the following format:

```
SEARCH list_name expression [,...]
       GET variable [,...]
       FLAG variable ;
```

The specified list is searched sequentially until an entry is found where each of the SEARCH expressions is equal in value to the corresponding LIST expression in the same expression list. The variables in the GET clause are then set to the corresponding remaining expressions of the expression list, and the FLAG variable is set to true. If no match is found, the variables specified in the GET clause are unchanged, and the FLAG variable is set to false.

5.9 Signalling

This section describes the signalling facilities of CFL.

Signalling breaks the control flow in the report to handle special conditions. Control goes to a special routine established by the user called a handler routine. When you signal a condition using the signalling statements, the most recently declared handler routine is called. The handler routine can then take the appropriate action.

Any routine can establish a handler routine. When you signal a condition, you can optionally suppress the change in the flow of control, and cause the handler to return to the routine executing the signal.

When a condition is signalled, a message describing the event is appended to the file ERROR, if the file exists. The message inserted in the ERROR file consists of a sequence of comma-delimited quoted strings, corresponding to the arguments to the SIGNAL-class statements.

5.9.1 The ENABLE Statement

The ENABLE statement has the following format:

```
ENABLE [MODULE expression] PROCEDURE expression ;
```

The procedure becomes the condition handler for this procedure and all called procedures, unless a called procedure in turn contains an ENABLE statement.

5.9.2 The SIGNAL Statement

The SIGNAL statement has the following format:

```
SIGNAL message_code PARAMETERS expression_list ;
```

The message code and expressions in the expression list are ASCII strings. When the SIGNAL statement is executed, these expressions are evaluated and the resulting ASCII strings are appended to the ERROR file as quoted strings separated by commas. The format is as follows:

```
'message code','expression1'[,'expression2'[,...]]
```

The signal-handling routine is then called. After execution of the signal-handling procedure, execution resumes following the statement.

5.9.3 The SIGNAL_STOP Statement

The SIGNAL_STOP statement is the same as the SIGNAL statement, except that after execution of the signal-handling procedure, execution resumes following the call to the procedure that executed the ENABLE statement. The format is as follows:

```
SIGNAL_STOP message_code PARAMETERS expression_list ;
```

5.9.4 The MESSAGE Statement

The MESSAGE statement has the same format as the SIGNAL statement. It causes the appended string to be placed in the ERROR file, but does not cause any signal processing. The format is as follows:

```
MESSAGE message_code PARAMETERS expression_list ;
```

5.9.5 The CRASH Statement

The CRASH statement causes an immediate abort of RPT. You use it in cases of error handling where the signalling mechanism is inadequate. The CRASH statement has the following format:

```
CRASH       ;
```

CRASH causes a detailed dump of many of RPT's internal data structures.

5.10 Print Formatting

This section describes the output-formatting facilities of CFL.

5.10.1 The FORMAT String

The FORMAT parameter in the WRITE and WRITE_GROUP statements expresses output-formatting information. The FORMAT parameter has the following syntax:

```
FORMAT format_string
```

The format string can be any ASCII string. The string is printed after substitution specified by output directives. The directives have the following format:

```
![n][m]dd
```

In this format, dd is the 2-character directive, m is the optional argument, and n is the optional repeat count. The square brackets need not be included if there is no repeat count. This syntax is the same as that for the VMS %FAO facility. A double exclamation point (!!) prints as a single exclamation point.

Multiple format strings can be specified by separating them with the concatenation operator, the vertical bar (|), which is ASCII 174. They are treated as one concatenated string.

The allowed directives are described in the following sections.

5.10.1.1 Control Directives

The control directives control the processing of the format string. The control directives are as follows:

!nCE	Repeat the FORMAT clause.
!nCF	When used with a WRITE statement, this directive terminates output if the values of all expressions have been output. When used with a WRITE_GROUP statement, this directive terminates output if all fields in the specified group have been output. The effect in both cases is to terminate evaluation and output of the format string if there are no more values to be output.

5.10.1.2 Formatting Directives

The formatting directives output carriage-control information. The formatting directives are as follows:

!nFC	Print following output beginning at column n.
!nFS	Space the current output print column forward n columns.
!nFL	Output n-1 blank lines. Printing resumes on the line following the blank lines. The default, n=1, causes output to begin on the line following the current line.
!nFP	Output a page break.

5.10.1.3 Data-Formatting Directives

The data-formatting directives control the output of data. The data-formatting directives are as follows:

!nDF — Print the field name of the current output field. The argument specifies the field width to be used for the name. The name is printed left-justified.

!nDP — Print the current output field.

The argument specifies the field width to be used for the field. For numeric fields, the field width n must be greater than the field width specified when the field was defined. If all fields have been printed, output terminates.

5.11 User-Interface Handling

The CFL compiler implements two user-interface modes: command mode and option mode. In command mode, you specify the input and output files to be processed, using either the MCR CFL command or the DCL CREATE/CFL command. In option mode, the compiler prompts you for option lines before compilation takes place.

You do not need to invoke the option mode if you are in the MCR command mode. The compiler automatically prompts you for option lines after you have entered your command line. In DCL, however, you must explicitly invoke option mode from the command line by using the /OPTIONS qualifier.

5.11.1 Command Mode

Format

```
CFL>output_file[,list_file][,symbol_file]=input_file[,symbol_file]

$ CREATE/CFL[/[NO]INTERMEDIATE_FORM][/LIST] input_file[/OPTIONS] [/SYMBOL]
```

In the MCR command line, all files to the left of the equal sign (=) are output files, and all files to the right of it are input files.

The MCR arguments and DCL qualifiers to the command mode are as follows:

output file
/[NO]INTERMEDIATE_FORM[:iform_file]

The output file is the compiled intermediate form (IFORM) output file. It has the default file type ICF. In MCR syntax, you must specify the output file.

The DCL CREATE/CFL command assumes that the output file has the same name as the input file, unless you use the /INTERMEDIATE_FORM qualifier to specify another name. The /NOINTERMEDIATE_FORM qualifier suppresses the creation of an IFORM (object) file.

listing file
/LIST[:list_file]

The compiler listing file has the default file name input_file.LST.

In DCL, you use the /LIST qualifier to generate a compiler listing file. The default is not to create a listing file.

symbol file
/SYMBOL[:symbol_file]

The symbol output file is a compilation symbol table. The symbol file must be specified as an argument whenever you compile a module that will be called from this module at execution time. It has the default file name input_file.SYM.

The symbol input file is the compilation symbol table from the module that calls the module being compiled at execution time.

In DCL, you use the /SYMBOL qualifier to create an output symbol file. The default is not to create a symbol file.

input file

The input file is the CNF source file.

/OPTIONS

In the DCL CREATE/CFL command line, you use the /OPTIONS qualifier to invoke the option mode of the CFL compiler. If you do not specify the /OPTIONS qualifier, the compiler does not implement the option mode, but proceeds to execute your command line.

In the MCR command line, there is no /OPTIONS qualifier. The compiler invokes the option mode before it executes your command line.

5.11.2 Option Mode

Option mode uses MCR only. When you enter the MCR CFL command line or the DCL command line with the /OPTIONS qualifier, the compiler requests options with the following option-mode prompt:

```
Option>
```

Press the RETURN key after each option you enter, and the Option> prompt reappears. Options are terminated when a line beginning with a slash (/) is entered, at which point the compilation takes place. See Section 4.2.3 for an example of CFL option-mode declarations.

In option mode, the compiler accepts the LITERAL option in the following format:

```
Option>LITERAL group.name = value
```

This is the same syntax as for the LITERAL statement in the source file. The LITERAL option is declared for the duration of the compilation. The only valid values are the following items:

- Quoted strings
- Numeric values
- Logical true values
- False values

Numeric values must be positive decimal values that are treated as machine values.

5.12 ERLCFL Report Messages

ERLCFL-F-ASCIIBIG, ASCII literal quoted string too long for type.

Explanation: An ASCII radix numeric literal in a control file source module contains too many characters for the specified numeric type.

User Action: Correct the user-written module or submit a Software Performance Report (SPR) for DIGITAL-supplied modules.

ERLCFL-F-BADDIGIT, Invalid numeric digit in conversion.

Explanation: A numeric literal or the ASCII string argument for the %COD$OCTAL, %COD$DECIMAL, %COD$HEX, %COD$BCD, %COD$BINARY, or %COD$MACHINE function contains an invalid character for the specified radix or was null or blank.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BITFLDSIZ, Bit or field too large in extraction operation.

Explanation: The bit or field in an extraction operation exceeds the size of the value on which the extraction is performed.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BITNOPREC, A BIT or FIELD must have a preceding data item.

Explanation: A BIT or FIELD declaration in a control file source module must be preceded by a data item within the declaration.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BITNOTVAR, A BIT or FIELD not allowed on variable-length data item.

Explanation: A BIT or FIELD declaration in a control file source module is not allowed on a variable-length data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BITNUMINV, BIT number outside the declared data item.

Explanation: The bit number in a BIT declaration for a data item is too large for the data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BITTOOHIG, Bit number too large for specified storage unit.

Explanation: The bit number specified by the character string portion of a #BI, #WI, #LI, #QI, or #VI numeric literal is too large for the specified value size.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-BYTERAD50, A BYTE data item cannot print in Radix-50 format.

Explanation: The print radix for a BYTE declaration cannot be Radix-50.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-CASENOTDAT, A declaration clause must be in a data declaration.

Explanation: A declaration CASE clause attempted to declare data but was not contained within a declaration.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-CFLINPUT, Could not open input source file.

Explanation: CFL could not open the input source file specified on the command line.

User Action: Check the file specification, and make sure that you have access to the specified file.

ERLCFL-F-CFLISTING, Could not create listing output file.

Explanation: CFL could not create the listing output file specified on the command line.

User Action: Check the file specification, and make sure that you have access to the specified file.

ERLCFL-F-CFLMODULE, Could not create module output file.

Explanation: CFL could not create the module output file specified on the command line.

User Action: Check the file specification, and make sure that you have access to the specified file.

ERLCFL-F-CFLSYMBOL, Could not open symbol file for input.

Explanation: CFL could not open the input symbol file specified on the command line.

User Action: Check the file specification, and make sure that you have access to the specified file.

ERLCFL-F-CFLSYMOUT, Could not create symbol output file.

Explanation: CFL could not create the output symbol file specified on the command line.

User Action: Check the file specification, and make sure that you have access to the specified file.

ERLCFL-F-CMDOPTERR, Option line syntax error.

Explanation: CFL encountered a syntax error on the option line input.

User Action: Correct the error and run CFL again, or submit an SPR for DIGITAL-supplied command files.

ERLCFL-F-CMDSPCERR, Command line syntax error.

Explanation: CFL encountered a syntax error on the command line input.

User Action: Correct the error and run CFL again, or submit an SPR for DIGITAL-supplied command files.

ERLCFL-F-DECTOOBIG, Declaration too large, too many symbols.

Explanation: A declaration in a control file is too large to be compiled.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-DIVZERO, Attempt to divide by zero.

Explanation: A control file module contains a division by zero in a compile-time constant expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-EXPTOOBIG, Operator stack overflow. Expression too complex.

Explanation: An expression in a control file source module is too complex to be compiled.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-EXPTOOLAR, Operator stack overflow. Expression too complex.

Explanation: An expression in a control file source module is too complex to be compiled.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FIELDBIG, FIELD exceeds size of the declared data item.

Explanation: A FIELD declaration in a control file source module exceeds the bounds of its corresponding data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FIELDBITI, FIELD starting bit is outside the declared data item.

Explanation: A FIELD declaration in a control file source module exceeds the bounds of its corresponding data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FIELDSMAL, FIELD width must be at least one bit.

Explanation: A FIELD declaration in a control file source module did not have a width specified.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FILERCLOS, File close error.

Explanation: An error occurred when CFL attempted to close a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLCFL-F-FILERREAD, File read error.

Explanation: An error occurred when CFL attempted to read a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLCFL-F-FILERWRIT, File write error.

Explanation: An error occurred when CFL attempted to write to a file.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLCFL-F-FILINTOPN, Internal error - File already open.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-FILINVCOD, Internal error - Invalid file code for specified operation.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-FLUSHINV, FLUSH attribute not allowed with KEEP attribute.

Explanation: A control file source module specified both the FLUSH and KEEP module attributes.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNFIELDS, Invalid conversion code argument to time conversion function.

Explanation: A control file source module contains a time conversion function with an invalid value for the conversion code argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNINVPOI, Invalid string pointer value in string function.

Explanation: A control file source module contains a %STR$PARSE or %STR$QUOTE function where the value of the pointer argument is larger than the length of the string argument.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNNOTCHA, Argument to STR$CHAR is not in valid range for character.

Explanation: The value of the argument for the %STR$CHAR function must be in the range 0 to 127_{10}.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNNOTCOM, Function call not allowed in compile-time constant expression.

Explanation: A control file source module contains a function call that could not be evaluated at compile time, where a compile-time constant expression was required.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNQUOODD, Quote string in STR$QUOTE function must have even length.

Explanation: A control file source module contains a %STR$QUOTE function, where the quote string argument is not an even length.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNSTRSIZ, Output string from string function too large.

Explanation: A control file source module executed a string function that resulted in a string longer than 255 characters.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNWRONGA, Incorrect number of arguments in function call.

Explanation: A control file source module contains a function call with the wrong number of arguments.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-FUNWRONGC, Incorrect number of arguments in function call.

Explanation: A control file source module contains a function call with the wrong number of arguments.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-HEAPOVERF, Heap too small to hold value. Overflow.

Explanation: The heap used for processing values and expressions has overflowed.

User Action: Edit CFLBLD.CMD to increase the extension for program section VHEAP0, and rebuild CFL.

ERLCFL-F-IFNOTDATA, A declaration IF clause must be in a data declaration.

Explanation: An IF clause cannot be used to declare data outside of a declaration in a control file source module.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-ILLCHAR, Illegal character in input.

Explanation: An invalid character was found in a control file source module.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-INTEOTMAN, Internal - More than one operator on stack at term end.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTEOTNUL, Internal - End of term reached with null operator stack.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTEXPNOO, Internal - Operator missing from operator stack.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTFUNEND, Internal - Stack entry missing at function termination.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTFUNMIS, Internal - Function code missing from operator stack.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTFUNNOT, Internal - Function code missing at function termination.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTOPRNOT, Internal - Operator outside of an expression term.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTPROUND, Internal - Compiler internal production stack underflow.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTSYMLNK, Internal error - Invalid symbol linkage setup in module.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INTWRONGP, Internal - Wrong production popped internal production.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INVFUNCT, Invalid function name specified.

Explanation: A control file source module specified an invalid function name.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-INVNUMSIZ, Internal - A numeric variable has an invalid size.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-INVPOIACT, Invalid POINTER-statement action name.

Explanation: A control file source module specified an invalid action for a POINTER statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-INVRADCNV, Internal error - Invalid radix code for conversion.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-KEEPINV, KEEP attribute not allowed with FLUSH attribute.

Explanation: A control file source module specified both the FLUSH and KEEP module attributes.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-LITINVTYP, Internal error - Literal in literal table has invalid type.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-LITNOVALU, Internal error - No value to load into literal value.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-MODATTRIN, Invalid module attribute name specified.

Explanation: A control file source module specified an invalid module attribute.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-MODZERO, Attempt to modulus by zero.

Explanation: A control file source module contains a modulus by zero in a compile-time constant expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-NOQUOTE, String literal missing closing quote.

Explanation: A string literal in a control file source module was not terminated by a closing apostrophe.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-NULLOPERA, Internal - Null suffix operand on non-suffix operator.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-NUMFILLCH, A print fill character string must contain one character.

Explanation: A print fill character in a declaration in a control file source module must contain one character.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-NUMINVOPR, Invalid numeric double-operand operation code.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-OPRFLSCOM, Internal error - Attempt to flush a CTCE operand.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-OPRINVLOG, Attempt to perform logical operation on an invalid type.

Explanation: A control file source module attempted to perform a logical operation with operands that were neither numeric nor logical.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-OPRNOTCOM, Operator in CTCE cannot be evaluated at compile-time.

Explanation: A compile-time constant expression in a control file source module contains an operator that could not be evaluated at compile time.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-OPRNOTIMP, Operation not implemented.

Explanation: A control file source module attempted to perform a multiplication where both operands were larger than a word value.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-PARSEREOF, Premature EOF encountered.

Explanation: The end-of-file was reached on a control file source module before the end of the module was reached.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-PROSTKOVR, Compiler internal production stack overflow.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-PRSSTKOVR, Parse stack overflow.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-RAD50BYTE, Cannot convert a byte using Radix-50 conversion.

Explanation: A control file source module attempted to convert an ASCII string or numeric literal to a BYTE using Radix-50 conversion.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-RADINVCOD, Invalid radix code string in radix literal.

Explanation: A control file source module contains an invalid radix code in a numeric literal.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-RADINVRAD, Invalid radix character specified in radix literal.

Explanation: A control file source module contains an invalid radix code in a numeric literal.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-RADLITINV, Invalid literal type character in radix literal.

Explanation: A control file source module contains an invalid type character in the radix portion of a numeric literal.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-RELINVCOD, Invalid relational operator.

Explanation: This is an internal error within CFL.

User Action: Please submit an SPR with any information you have.

ERLCFL-F-RESBITILL, A BIT or FIELD data item cannot be declared RESERVED.

Explanation: A control file source module contains a BIT or FIELD declaration for a RESERVED data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-RESINVPRT, Print expression list not allowed on RESERVED data.

Explanation: A control file source module contains a print expression for a RESERVED data item.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-SPCNOTVAR, Special variable cannot be used in a CTCE.

Explanation: A control file source module used one of the predeclared special variables in a compile-time constant expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-SRCDATERR, Failed to get creation date of source file.

Explanation: An error occurred when CFL tried to get the creation date of the control file source module.

User Action: Check for file access conflicts, device errors, or low pool condition.

ERLCFL-F-SUBEXTBIG, Substring extraction end element exceeds string.

Explanation: A control file source module attempted to perform a substring extraction in which the substring exceeded the end of the string.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-SYMNOTCOM, A variable is not valid in a compile-time constant expression.

Explanation: A control file source module contains a variable in a compile-time constant expression.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-SYMNOTLIT, Specified LITERAL symbol name not part of LITERAL group.

Explanation: A control file source module contains a reference to a LITERAL symbol that has not been defined for the specified LITERAL group.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-SYNTAXERR, Syntax error.

Explanation: CFL encountered a syntax error while compiling the control file source module.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-TABLEBIG, TABLE element has too many literal values.

Explanation: A control file source module contains a TABLE element with too many values for the corresponding TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-TABLESMAL, TABLE element has too few literal values.

Explanation: A control file source module contains a TABLE element with too few values for the corresponding TABLE.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-VALSTKOVR, Value stack overflow.

Explanation: The stack used for processing values and expressions has overflowed.

User Action: Edit CFLBLD.CMD to increase the extension for program section VLSTK0, and rebuild CFL.

ERLCFL-F-VALUESIZE, Value in expression is too large.

Explanation: A control file source module contains a compile-time constant expression in which an intermediate value or the final value is too large.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-VALUETYPE, Value in expression is wrong type.

Explanation: A control file source module contains a compile-time constant expression in which an intermediate value or the final value is of the wrong type.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-VARLITGRP, A variable name cannot have the same group name as a LITERAL.

Explanation: A control file source module contains a declaration with the same group name as a LITERAL declaration.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

ERLCFL-F-WRITEDES, WRITE-class statement has invalid destination.

Explanation: A control file source module contains an invalid destination for the TO clause of a WRITE or WRITE_GROUP statement.

User Action: Correct the user-written module or submit an SPR for DIGITAL-supplied modules.

Appendix A
Tuning the Error Logging Universal Library

When the Report Generator task (RPT) starts operation, it scans ERRLOG.ULB and generates descriptors in its work space for all the modules it finds in the library. RPT searches these descriptors for the appropriate modules when it generates reports. Therefore, if you remove modules for devices that your system does not have, RPT operation is faster and does not require as much memory or disk space.

An indirect command file, TUNE.CMD, is included with your error logging system. The command file is in the universal library LB:[1,6]ERRLOGETC.ULB. This command file allows you to remove unneeded modules from ERRLOG.ULB.

A.1 Using the TUNE Command File

To use TUNE.CMD, you must first extract it from the library by using the following LBR command:

```
>LBR TUNE.CMD=LB:[1,6]ERRLOGETC.ULB/EX:TUNE
```

The file TUNE.CMD is copied to your current directory. You can execute it using the following command:

```
@TUNE
```

The TUNE.CMD file prompts you for a processor type and for a list of the devices you want to use with error logging. It then removes the devices you do not select from ERRLOG.ULB and writes a new ERRLOG.ULB to the current directory.

TUNE.CMD requires the actual physical names of the devices you select (RL01 and RL02, for example). Since some modules in ERRLOG.ULB handle more than one physical device type, the tuning command includes the module only once. If you select a device name more than once, the system displays the following message on your terminal:

```
Module <module name> replaced.
```

The message is for your information only. The command file then continues to include the other modules you specified.

Example A–1 is an example of using TUNE.CMD.

This file creates a library for a PDP-11/84 processor with the following devices:

- RP06
- RP05
- RM03
- RM05
- RM80
- RS04
- RK05
- RK06
- RK07
- TU56
- RX02
- TU58

Example A–1: Sample Execution of TUNE.CMD

```
$ @TUNE [RET]

;       This command procedure is used to tune the error log control file for
;       your system configuration.  The procedure prompts for the location of
;       the  master  control  file (the  shipped file), the CPU type, and the
;       error logging devices available on your system. It then creates a new
;       version  of the control file that contains only the required support.
;       The original control file is left unchanged.

;       All  files  are  created  in  your  default  directory on the default
;       device.  When  cleaning  up,  all files with the extension of .ICF as
;       well as TEMPTUNE0.TMP;* and TEMPTUNE1.TMP;* are deleted.

* Continue? [Y/N]: Y [RET]

* Location of master file [D: LB:[1,6]ERRLOG.ULB] [S]: [RET]

;       Now  enter  the  CPU type.  Hit the escape key for a list of legal
;       CPU types.
;
* Enter CPU type [S]: [RET]
;
;       The acceptable CPU types are:
;       11/23,  11/24,  11/44
;       11/70,  11/74
;       11/83,  11/84,  11/53,  11/73

* Enter CPU type [S]: 11/70 [RET]
;

;       Now  enter  the  devices in your configuration separated by commas.
;       Terminate  by  entering a period.  Hit the escape key for a list of
;       acceptable device names.

* Enter device name(s) [S]: [ESC]
;

;       Below  is a list of acceptable device names.  If you have more than
;       one type of device listed as "x or y or z" you need enter only one.
;       For  example,  if you have RP04s and RP06s you need only enter RP04
;       or RP06 - not both.

;       The acceptable device names are:

:       TU56 (DECtape)
;       TU58 (DECtape II)
;       TU60 (Cassette)
;       RP04 or RP05 or RP06
;       RP07
;       RS11
;       RK03 or RK05
;       RL01 or RL02
;       RK06 or RK07
;       RP02 or RP03
;       RM02 or RM03
;       RM05
```

(Continued on next page)

Example A–1 (Cont.): Sample Execution of TUNE.CMD

```
;       RM80
;       RS03 or RS04
;       RA60
;       RA80 or RA81
;       RC25
;       RD51 or RD52 or RD53 or RD54 or RD31 or RD32 or RX50 or RX33
;       RX01
;       RX02
;       ML11
;       TE16 or TU16 or TU45
;       TU77
;       TE10 or TU10 or TS03
;       TU78
;       TS11
;       TU80
;       TSV05
;       TK50 or TU81
* Enter device name(s) [S]: RP06,RP05,RM03,RM05,RM80 [RET]
* Enter device name(s) [S]: RS04,RK05,RK06,RK07,TU56,RX02,TU58 [RET]
* Enter device name(s) [S]: . [RET]
;
;       Extract the files from the master library.
;
;       Build  the  new library.  Note that you may see messages like "Module
;       "XYZZY" replaced" if  you have selected more than one device having
;       the same mnemonic.  For example, selecting RK06 and RK07 support will
;       produce this message. This type of message can be ignored.
;
LBR @TEMPTUNE1.TMP
Module "ERP456" replaced
Module "ERP456" replaced
Module "ERP456" replaced
Module "ERM23" replaced
Module "NRM23" replaced
Module "ERM05" replaced
Module "ERM80" replaced
Module "ERK67" replaced
Module "NRK67" replaced
;       Now clean up.
;
LBR TEMPLIB.ULB/CO=TEMLIB.ULB
PIP TEMPLIB.ULB/PU
PIP TEMPLIB.ULB/TR
PIP ERRLOG.ULB/RE/NV=TEMPLIB.ULB
PIP *.ICF;*,TEMPTUNE0.TMP;*,TEMPTUNE1.TMP;*/DE
;
;       Finished.
;
```

Now you can copy the new library to LB:[1,6], where it will become the default library for RPT. You should maintain the original DIGITAL-supplied ERRLOG.ULB, either on LB:[1,6] or in another location. You can then use TUNE.CMD again later on the original ULB file to add support for devices that you have taken out.

A.2 DIGITAL-Supplied Error Logging Modules

To list the modules in the DIGITAL-supplied universal library, use the following LBR command:

```
>LBR ERRLOG.ULB/LI RET
```

Table A–1 lists the modules in this library.

Table A–1: Modules in ERRLOG.ULB

Module Name	Module Description
DEVSM1	Defines tables for device modules
DEVUDA	MSCP-related module
DISPATCH	Entry module for error log control file
DMPALL	Processes errors from unknown events
DSP1P1	Processes Error Logger command packets
DSP2P1	Processes device error packets
DSP3P1	Processes device information packets
DSP4P1	Processes device control information
DSP5P1	Processes CPU memory-detected errors
DSP6P1	Processes system control information
DSP7P1	Processes control information packets
DSP8P1	Processes CPU-detected error packets

Table A–1 (Cont.): Modules in ERRLOG.ULB

Module Name	Module Description
EML11 ERK05 ERK67 ERL12 ERM05 ERM23 ERM80 ERP07 ERP23 ERP456 ERS11 ERS34 ERX01 ERX02 ETA11 ETC11 ETS11 ETU58 ETU77 ET0310 ET1645 ETSV05 ETK50 ETU81 ETK70	Control file device modules
ERRORM	Processes control file error conditions
EUNKWN	Processes errors from unknown devices
E11XX E1144 E117X E118X	Process CPU/memory packets
FINLP1	Cleans up control file after processing
FMTNP1	Prints NARROW reports
FMTWP1	Prints WIDE reports
INITM1	Initializes control file modules
MSCPCE MSCPAT MSCP80 MSCP60 MSCPEN MSCPTO MSCPSD MSCP5X	MSCP device modules

Table A–1 (Cont.): Modules in ERRLOG.ULB

Module Name	Module Description
NML11 NRK67 NRM05 NRM23 NT0310 NTS11	Generate notes from device modules
PARSEM PARS1M PARS2M PARS3M PRS2AM PRS2BM	RPT command line parser
SELTM1	Selects packets to process
SMRYEP	Prints ERROR summary report
SMRYGP	Prints GEOMETRY summary report
SMRYHP	Prints HISTORY summary report
TMSCPE	TMSCP device module

Appendix B
Drive Serial Numbers

RPT reports drive serial numbers for those devices that have serial numbers. Table B–1 lists the drives that provide serial numbers and the significant digits RPT uses from those serial numbers. The number of digits varies by drive type. The digits appear in binary coded decimal (BCD) format.

RPT obtains the serial number from the device electronics. The number is selected through a series of jumpers within the device that are set at manufacturing time. These jumpers match the low-order digits of the actual device serial number. If any of these jumpers is altered, the drive will have a different serial number unless the new jumpers are set to reflect the actual device serial number. Therefore, the number RPT prints may not be the same as the number on the device identification plate.

Table B–1: Significant Digits in Drive Serial Numbers

Device/Controller	Significant Digits in Serial Number
RK06	12 bits, 3 BCD digits
RK07	12 bits, 3 BCD digits
RM80	16 bits, 4 BCD digits
TU77	16 bits, 4 BCD digits
TU78	16 bits, 4 BCD digits
RP04	16 bits, 4 BCD digits
RP05	16 bits, 4 BCD digits
RP06	16 bits, 4 BCD digits
RP07	16 bits, 4 BCD digits
RX01	N/A
RX02	N/A

Table B–1 (Cont.): Significant Digits in Drive Serial Numbers

Device/Controller	Significant Digits in Serial Number
RM02	16 bits, 4 BCD digits
RM03	16 bits, 4 BCD digits
RM05	16 bits, 4 BCD digits
TA11	N/A
RS11	N/A
RP02	N/A
RP03	N/A
ML11	16 bits, 4 BCD digits
TE16/RH11/RH70	16 bits, 4 BCD digits
TU16/RH11/RH70	16 bits, 4 BCD digits
TU45	16 bits, 4 BCD digits
TS03	N/A
TE10/TMB11	N/A
TU10/TMB11	N/A
RS03	N/A
RS04	N/A
TS11	N/A
TC11	N/A
RL01	N/A
RL02	N/A
TSV05	N/A
RA60/RA80/RA81/RA82	6 digits
RC25/RD31/RD32/RX50/RX33	6 digits
RD51/RD52/RD53/RD54	6 digits
TU80	N/A
TU81E	N/A
TK50/TK70	N/A

Appendix C
Error Log Packet Format

Example C–1 shows the format of an error log packet in memory, as described in the system macro EPKDF.MAC.

When a device error is logged, the error log packet contains device-supplied information to describe the error. This information usually consists of the device registers and some additional information supplied by the system.

Error logging also writes context information into the error log packet. This information includes the time and date of the error, information about the system that logged the error, and information about the I/O operation that generated the error.

The error logging system also creates packets for events in the system that are not errors, but are important to the interpretation of errors (for example, the time error logging starts or stops).

Example C–1: Error Log Packet Format

```
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
; ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
; COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
; OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
; TRANSFERRED.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
; AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
; CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.

;       IDENT /V02.00/
;
; Previously modified by:
;
;       C. Putnam
;
; Modified By:
;       C. Putnam       - 27-MAY-83     1.04
;                       - CPU Detected Errors become code 8
;                       - MEMORY Detected Errors only are code 5
;
;       S. Adams        - 10-SEPT-83    1.05
;                       - ADD BIT TO INDICATE LIMIT REACHED
;
;       J. MELVIN       - 06-NOV-84     V02.00
;          JJM100       - DEFINE ERROR LOG FEATURE MASK DEFINITIONS
;       G. Marigowda    - 17-APR-85     2.01
;                       - Change E$ORTY to two bytes instead of one word
;
;       G. Marigowda    - 25-APR-85     2.02
;                       - Add new flag bits in E$OFLG and E$AFLG to indicate
;                       - internal I/O operation.
;       G. Marigowda    - 17-MAY-85     2.03
;                         Change Current packet format from 1 to 2.
;
      .MACRO   EPKDF$,L,B
;+
; Error Message Block Definitions
;-
      .ASECT
;
; Header Subpacket
;
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
;       +-----------------------------------------------+
;       | Subpacket Length in Bytes                     |
;       +-----------------------------------------------+
;       | Subpacket Flags                               |
;       +-----------------------+-----------------------+
;       | Format Identification | Operating System Code |
;       +-----------------------+-----------------------+
;       | Operating System Identification               |
;       |                                               |
;       +-----------------------+-----------------------+
;       | Flags                 | Context Code          |
;       +-----------------------+-----------------------+
;       | Entry Sequence                                |
;       +-----------------------------------------------+
;       | Error Sequence                                |
;       +-----------------------+-----------------------+
;       | Entry Type Subcode    | Entry Type Code       |
;       +-----------------------+-----------------------+
;       | Time Stamp                                    |
;       |                                               |
;       |                                               |
;       +-----------------------+-----------------------+
;       | Reserved              | Processor Type        |
;       +-----------------------+-----------------------+
;       | Processor Identification (URM)                |
;       +-----------------------------------------------+

.=0

E$HLGH:'L'    .BLKW   1    ; Subpacket Length in Bytes
E$HSBF:'L'    .BLKW   1    ; Subpacket Flags
E$HSYS:'L'    .BLKB   1    ; Operating System Code
E$HIDN:'L'    .BLKB   1    ; Format Identification
E$HSID:'L'    .BLKB   4    ; Operating System Identification
E$HCTX:'L'    .BLKB   1    ; Context Code
E$HFLG:'L'    .BLKB   1    ; Flags
E$HENS:'L'    .BLKW   1    ; Entry Sequence Number
E$HERS:'L'    .BLKW   1    ; Error Sequence Number
E$HENC:'L'                 ; Entry Code
E$HTYC:'L'    .BLKB   1    ; Entry Type Code
E$HTYS:'L'    .BLKB   1    ; Entry Type Subcode
E$HTIM:'L'    .BLKB   6    ; Time Stamp
E$HPTY:'L'    .BLKB   1    ; Processor Type
              .BLKB   1    ; Reserved

E$HURM:'L'    .BLKW   1    ; Processor Identification (URM)
              .EVEN
E$HLEN:'L'                 ; Length
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
; Subpacket Flags for E$HSBF
        SM.ERR   ='B'        1   ; Error Packet
        SM.HDR   ='B'        1   ; Header Subpacket
        SM.TSK   ='B'        2   ; Task Subpacket
        SM.DID   ='B'        4   ; Device Identification Subpacket
        SM.DOP   ='B'       10   ; Device Operation Subpacket
        SM.DAC   ='B'       20   ; Device Activity Subpacket
        SM.DAT   ='B'       40   ; Data Subpacket
        SM.MBC   ='B'    20000   ; 22-bit MASSBUS controller present
        SM.CMD   ='B'    40000   ; Error Log Command Packet
        SM.ZER   ='B'   100000   ; Zero I/O Counts

; Codes for field E$HIDN
;
        EH$FOR   ='B'        1   ; Current packet format
;
; Flags for the error log flags byte ($ERFLA) in the Exec.

        ES.INI   ='B'        1   ; Error log initialized
        ES.DAT   ='B'        2   ; Error log receiving data packets
        ES.LIM   ='B'        4   ; Error limiting enabled
        ES.LOG   ='B'       10   ; Error logging enabled

; Type and Subtype Codes for fields E$HTYC and E$HTYS

;       Symbols with names E$Cxxx are type codes for field E$HTYC,
;       Symbols with names E$Sxxx are subtype codes for field E$HTYS.

        E$CCMD    ='B'      1  ;  Error Log Control
        E$SSTA    ='B'      1  ;       Error Log Status Change
        E$SSWI    ='B'      2  ;       Switch Logging Files
        E$SAPP    ='B'      3          Append File
        E$SBAC    ='B'      4  ;       Declare Backup File
        E$SSHO    ='B'      5  ;       Show
        E$SCHL    ='B'      6  ;       Change Limits

        E$CERR    ='B'      2  ;  Device Errors
        E$SDVH    ='B'      1  ;       Device Hard Error
        E$SDVS    ='B'      2  ;       Device Soft Error
        E$STMO    ='B'      3  ;       Device Interrupt Timeout (HARD)
        E$SUNS    ='B'      4  ;       Device Unsolicited Interrupt
        E$STMS    ='B'      5  ;       Device Interrupt Timeout (SOFT)

        E$CDVI    ='B'      3  ;  Device Information
        E$SDVI    ='B'      1  ;       Device Information Message
        E$CDCI    ='B'      4  ;  Device Control Information
        E$SMOU    ='B'      1  ;       Device Mount
        E$SDMO    ='B'      2  ;       Device Dismount
        E$SRES    ='B'      3  ;       Device Count Reset
        E$SRCT    ='B'      4  ;       Block Replacement
        E$CMEM    ='B'      5  ;  Memory Detected Errors
        E$SMEM    ='B'      1  ;       Memory Error

        E$CSYS    ='B'      6  ;  System Control Information
        E$SPWR    ='B'      1  ;       Power Recovery
        E$CCTL    ='B'      7  ;  Control Information
        E$STIM    ='B'      1  ;       Time Change
        E$SCRS    ='B'      2  ;       System Crash
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
        E$SLOA    ='B'      3  ;       Device Driver Load
        E$SUNL    ='B'      4  ;       Device Driver Unload
        E$SHRC    ='B'      5  ;       Reconfiguration Status Change
        E$SMES    ='B'      6  ;       Message
        E$CCPU    ='B'     10  ;  CPU Detected Errors
        E$SINT    ='B'      1  ;       Unexpected Interrupt
;       E$SINT    ='B'      2  ;       Unexpected Interrupt
; Subtype code 2 is reserved.  Use 3 for the next following Subtype code

        E$CSDE    ='B'     11  ;  Software Detected Events
        E$SABO    ='B'      1  ;        Task Abort

; Codes for Context Code entry E$HCTX
;
        EH$NOR    ='B'      1  ;  Normal Entry
        EH$STA    ='B'      2  ;  Start Entry
        EH$CRS    ='B'      3  ;  Crash Entry

; Codes for Flags entry E$HFLG
;
        EH$VIR    ='B'      1  ;  Addresses are virtual
        EH$EXT    ='B'      2  ;  Addresses are extended
        EH$COU    ='B'      4  ;  Error counts supplied
        EH$QBS    ='B'     10  ;  Q-BUS CPU

; Task Subpacket
;
;       +-----------------------------------------------+
;       | Task Subpacket Length                         |
;       +-----------------------------------------------+
;       | Task Name in RAD50                            |
;       +-----------------------------------------------+
;       | Task UIC                                      |
;       +-----------------------------------------------+
;       | Task TI: Device Name                          |
;       +-----------------------+-----------------------+
;       | Flags                 | Task TI: Unit Number  |
;       +-----------------------+-----------------------+
.=0

E$TLGH:'L'    .BLKW   1    ; Task Subpacket Length
E$TTSK:'L'    .BLKW   2    ; Task Name in RAD50
E$TUIC:'L'    .BLKW   1    ; Task UIC
E$TTID:'L'    .BLKB   2    ; Task TI: Device Name
E$TTIU:'L'    .BLKB   1    ; Task TI: Unit
E$TFLG:'L'    .BLKB   1    ; Flags

              .EVEN
E$TLEN:'L'
;
; Flags for entry E$TFLG

        ET$PRV    ='B'      1  ;  Task is Privileged
        ET$PRI    ='B'      2  ;  Terminal is Privileged
;
; Device Identification Subpacket
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
;       +-----------------------------------------------+
;       | Device Identification Subpacket Length        |
;       +-----------------------------------------------+
;       | Device Mnemonic Name                          |
;       +-----------------------+-----------------------+
;       | Controller Number     | Device Unit Number    |
;       +-----------------------+-----------------------+
;       | Physical Subunit #    | Physical Unit #       |
;       +-----------------------+-----------------------+
;       | Physical Device Mnemonic                      |
;       +-----------------------+-----------------------+
;       | Reserved              | Flags                 |
;       +-----------------------+-----------------------+
;       | Volume Name of Mounted Volume                 |
;       +-----------------------------------------------+
;       | Pack Identification                           |
;       +-----------------------------------------------+
;       | Device Type Class                             |
;       +-----------------------------------------------+
;       | Device Type                                   |
;       +-----------------------------------------------+
;       | I/O Operation Count Longword                  |
;       +-----------------------+-----------------------+
;       | Hard Error Count      | Soft Error Count      |
;       +-----------------------+-----------------------+
;       | Blocks Transferred Count                      |
;       +-----------------------------------------------+
;       | Cylinders Crossed Count                       |
;       +-----------------------------------------------+

.=0
E$ILGH:'L'    .BLKW   1    ; Device Identification Subpacket Length
E$ILDV:'L'    .BLKW   1    ; Device Mnemonic Name
E$ILUN:'L'    .BLKB   1    ; Device Unit Number
E$IPCO:'L'    .BLKB   1    ; Controller Number
E$IPUN:'L'    .BLKB   1    ; Physical Unit Number
E$IPSU:'L'    .BLKB   1    ; Physical Subunit Number

              .IF  DF R$$MPL

E$IPDV:'L'    .BLKW   1    ; Physical Device Mnemonic

              .ENDC        ; R$$MPL
E$IFLG:'L'    .BLKB   1    ; Flags
              .BLKB   1    ; Reserved
E$IVOL:'L'    .BLKB   12.  ; Volume Name

E$IPAK:'L'    .BLKB   4    ; Pack Identification
E$IDEV:'L'                 ; Device Type
E$IDCL:'L'    .BLKW   1    ; Device Type Class
E$IDTY:'L'    .BLKW   2    ; Device Type
E$IOPR:'L'    .BLKW   2    ; I/O Operation Count Longword
E$IERS:'L'    .BLKB   1    ; Soft Error Count
E$IERH:'L'    .BLKB   1    ; Hard Error Count
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
            .IF  DF R$$MPL

E$IBLK:'L'  .BLKW   2   ; Blocks transferred count
E$ICYL:'L'  .BLKW   2   ; Cylinders crossed count

            .ENDC       ; R$$MPL

            .EVEN
E$ILEN:'L'              ; Subpacket Length
;
; Flags for field E$IFLG

   EI$SUB   ='B'        1   ; Subcontroller device
            .IF  DF R$$MPL

   EI$NUX   ='B'        2   ; No UCB extension, data invalid
            .ENDC ; R$$MPL
;
; Device Operation Subpacket
;

;      +-----------------------------------------------+
;      | Device Operation Subpacket Length             |
;      +-----------------------------------------------+
;      | Task Name in RAD50                            |
;      +-----------------------------------------------+
;      | Task UIC                                      |
;      +-----------------------------------------------+
;      | Task TI: Logical Device Mnemonic              |
;      +----------------------+------------------------+
;      | Reserved             | Task TI: Device Unit   |
;      +----------------------+------------------------+
;      | I/O Function Code                             |
;      +----------------------+------------------------+
;      | Reserved             | Operation Flags        |
;      +----------------------+------------------------+
;      | Transfer Operation Address                    |
;      +-----------------------------------------------+
;      | Transfer Operation Byte Count                 |
;      +-----------------------------------------------+
;      | Current Operation Retry Count                 |
;      +-----------------------------------------------+
;

.=0
E$OLGN:'L'  .BLKW   1   ; Subpacket Length
E$OTSK:'L'  .BLKW   2   ; Task Name in RAD50
E$OUIC:'L'  .BLKW   1   ; Task UIC
E$OTID:'L'  .BLKB   2   ; Task TI: Logical Device Mnemonic
E$OTIU:'L'  .BLKB   1   ; Task TI: Logical Device Unit
            .BLKB   1   ; Reserved

E$OFNC:'L'  .BLKW   1   ; I/O Function Code
E$OFLG:'L'  .BLKB   1   ; Operation Flags
            .BLKB   1   ; Reserved
E$OADD:'L'  .BLKW   2   ; Transfer Operation Address
E$OSIZ:'L'  .BLKW   1   ; Transfer Operation Byte Count
E$ORTY:'L'  .BLKW   1   ; Current Operation Retry Count
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
            .EVEN
E$OLEN:'L'                ; Device Operation Subpacket Length
;
; Flags for field E$OFLG

        EO$TRA    ='B'        1    ; Transfer Operation
        EO$DMA    ='B'        2    ; DMA Device
        EO$EXT    ='B'        4    ; Extended Addressing Device
        EO$PIP    ='B'       10    ; Device is positioning
;
; I/O Activity Subpacket

;       +-----------------------------------------------+
;       | I/O Activity Subpacket Length                 |
;       +-----------------------------------------------+
.=0

E$ALGH:'L'    .BLKW   1    ; Subpacket Length
;
; I/O Activity Subpacket Entry

;       +-----------------------------------------------+
;       | Logical Device Name Mnemonic                  |
;       +-----------------------+-----------------------+
;       | Controller Number     | Logical Device Unit   |
;       +-----------------------+-----------------------+
;       | Physical Subunit #    | Physical Unit Number  |
;       +-----------------------+-----------------------+
;       | Physical Device Mnemonic                      |
;       +-----------------------+-----------------------+
;       | Task TI: logical unit | Device flags          |
;       +-----------------------+-----------------------+
;       | Requesting Task Name in RAD50                 |
;       +-----------------------------------------------+
;       | Requesting Task UIC                           |
;       +-----------------------------------------------+
;       | Task TI: Logical Device Name                  |
;       +-----------------------------------------------+
;       | I/O Function Code                             |
;       +-----------------------+-----------------------+
;       | Reserved              | Flags                 |
;       +-----------------------+-----------------------+
;       | Transfer Operation Address                    |
;       +-----------------------------------------------+
;       | Transfer Operation Byte Count                 |
;       +-----------------------------------------------+
;

.=0
E$ALDV:'L'    .BLKW   1    ; Logical Device Name Mnemonic
E$ALUN:'L'    .BLKB   1    ; Logical Device Unit
E$APCO:'L'    .BLKB   1    ; Controller Number
E$APUN:'L'    .BLKB   1    ; Physical Unit Number
E$APSU:'L'    .BLKB   1    ; Physical Subunit Number
```

(Continued on next page)

Example C–1 (Cont.): Error Log Packet Format

```
            .IF  DF R$$MPL
E$APDV:'L'  .BLKW   1   ; Physical Device Mnemonic
            .ENDC
E$ADFG:'L'  .BLKB   1   ; Device flags
E$ATIU:'L'  .BLKB   1   ; Task TI: Logical Unit
E$ATSK:'L'  .BLKW   2   ; Requesting Task Name in RAD50
E$AUIC:'L'  .BLKW   1   ; Requesting Task UIC
E$ATID:'L'  .BLKW   1   ; Task TI: Logical Device Name
E$AFNC:'L'  .BLKW   1   ; I/O Function Code
E$AFLG:'L'  .BLKB   1   ; Flags
            .BLKB   1   ; Reserved
E$AADD:'L'  .BLKW   2   ; Transfer Operation Address
E$ASIZ:'L'  .BLKW   1   ; Transfer Operation Byte Count
            .EVEN
E$ALEN:'L'          ; Subpacket Entry Length
;
; Flags for field E$ADFG
        EA$SUB  ='B'        1   ; Subcontroller device
                .IF  DF R$$MPL
        EA$NUX  ='B'        2   ; No UCB extension, data invalid
                .ENDC ; R$$MPL
;
; Flags for field E$AFLG
        EA$TRA  ='B'        1   ; Transfer Operation
        EA$DMA  ='B'        2   ; DMA Device
        EA$EXT  ='B'        4   ; Device has Extended Addressing
        EA$PIP  ='B'       10   ; Device is positioning
        .PSECT
        .MACRO  EPKDF$  X,Y
        .ENDM
        .ENDM
        .IIF NDF S$$YDF , .LIST
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

RSX–11M–PLUS and Micro/RSX
Error Logging Manual
AA–JS19A–TC

READER'S COMMENTS

Your comments and suggestions are welcome and will help us in our continuous effort to improve the quality and usefulness of our documentation and software.

Remember, the system includes information that you read on your terminal: help files, error messages, prompts, and so on. Please let us know if you have comments about this information, too.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

What kind of user are you? ___ Programmer ___ Nonprogrammer

Years of experience as a computer programmer/user: _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

--- Do Not Tear - Fold Here and Tape ---

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---