PACIFIC ELECTRO DATA
PED-4010
INITIATOR EMULATION PROGRAM
USER'S MANUAL

PACIFIC ELECTRO DATA, INC.
14 Hughes, Suite B205, Irvine, California 92718
(714) 770-3244

SOFTWARE LICENSE

1. COPYRIGHT: This software is protected by both United States Copyright laws and International Treaty provisions.

2. LICENSE: The license is granted to authorize the Buyer, on a non-exclusive basis, to use each test program on each particular designated Data Acquisition and Emulation Module. The license is non-transferable.

3. COPIES: The buyer may make one (1) copy of each licensed test program for use on a particular designated Data Acquisition and Emulation Module for back-up purposes only.

4. PROTECTION OF LICENSE PROGRAM: The buyer agrees not to provide, or otherwise make available, any licensed test program, in any form, to any person other than Buyer and Buyer's premises with Buyer's permission for purposes specifically related to Buyer's user of the licensed test program. The Buyer agrees to take appropriate action by instruction, agreement, or otherwise with Buyer's employees or other persons permitted access to licensed test programs to satisfy his obligation under this license with respect to use, copying, protection, and security of licensed test programs.

SOFTWARE WARRANTY

1. WARRANTY: Pacific Electro Data, Inc. ("PED") warrants the physical diskette and physical documentation enclosed herein to be free of defects in materials and workmanship for a period of 90 days from the date of purchase. PED further warrants that the software conforms to all current specifications and samples for a period of one (1) year from the date of purchase. PED reserves the right to make changes in the software described herein without notice. At Buyer's request, PED will replace the obsolete software diskette and documentation during the Software Warranty period. All software diskettes replaced hereunder shall become the property of PED.

2. LIMITATIONS OF LIABILITY:

   A. The previous express warranty is the only warranty made by PED. PED grants no implied warranties, including warranties or merchantability or fitness and no other express warranties. The express obligation stated above is in lieu of all liabilities or obligations of PED for damages including, but not limited to, consequential damages arising out of or in connection with the delivery, use, or performance of PED test devices.

   B. Buyer further agrees that PED will not be liable for any lost profits or for any claim or demand against the buyer by any other party, except a claim for patent, copyright, and trademark infringement as provided herein.

   C. In no event will PED be liable for special, indirect, or consequential damages even if PED has been advised of the possibility of such damages. The risk of loss or damage for any products supplied by buyer to PED will be borne exclusively by buyer.

Table of Contents

Table of Contents (continued).

Figures

Tables

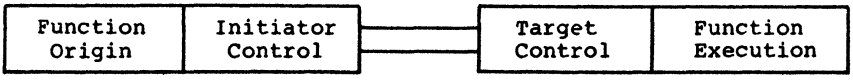
1.1 General Information

The PED-4010 Initiator Emulation Program is part of the PED-4000 Emulation and Analysis System. This package enables the IBM PC or compatible to emulate and analyze the Small Computer System Interface (SCSI).

The basic PED-4000 System consists of a PED-4001 Data Acquisition and Emulation Mode, and the State Analysis program. The module occupies a single, full-length expansion slot in the host computer and connects to the SCSI bus through a 50-pin connector. The programs the module uses can include the following:

- o PED-4001 State Analysis Program
- o PED-4002 Phase Analysis Program
- o PED-4010 Initiator Emulation Program
- o PED-4020 Target Emulation Program

As part of the complete package, the PED-4010 Initiator Emulation Program emulates the initiator (or source) device, and manipulates the target device using a variety of commands, definitions, and options. In this way, the user can analyze the function of the initiator and its interface with one or more target devices.



The PED-4010 Initiator Emulation program is an extraordinarily flexible programming environment, providing the user with great latitude in planning and executing initiation simulations. All operations are conducted through easy-to-use menus. Programmed and interactive emulations use a language employing common commands and syntax, facilitating writing and emulation processing. Moreover, all programs in the PED-4000 package use a uniform set of keyboard and display conventions, setup and operation menus, function key responses, and common service utilities, such as Help messages, File Load, File Save, and Print.

1.2 Overview of This Manual

This manual describes the installation, operation, and user/program interface procedures for using the PED-4010 Initiator Emulation Program with the PED-4001 module. It takes you through the entire process of defining the necessary host environments, writing and editing the programs for batch mode emulation, arranging and identifying command language for interactive mode emulation, planning strategies for initiator emulation.

It is designed as a stand-alone manual, requiring only limited reference to the PED-4001 State Analyzer Program User's Manual.

1.3 How To Use This Manual

This section tells you how to use this manual. If you are not familiar with the PED-4000 system, we recommend that you read the entire manual from beginning to end. If you have used some of the programs before, then you might select only those sections which directly concern you. This manual is set out in the following manner.

Section 1 presents an overview of the PED-4000 System and discusses the PED-4010 Initiator Emulation Program. In addition, it identifies the hardware and software required to run the initiator, and reviews installation and configuration procedures.

Section 2 describes the major resources and functional modules of the PED-4010 program, what they do, and how they interact during initiator emulation. Recommendations are then provided for the best tactics to follow in creating an emulation, depending on your particular requirements. Finally, the procedure for booting and bringing up the emulation is detailed.

Section 3 describes the user/program interface conventions. These include such keyboard operations as function and action keys, cursor movement, and alphanumeric character entry. Menu conventions are then discussed, including the window and prompt structure.

Section 4 tells how to define the target including the use of the Device Description Library (DDL) and the defining and creating of both Command Descriptor Blocks (CDB) and Data Buffers.

Section 5 describes how to configure a DDL for device assignment, SCSI addresses, and driver options.

Section 6 tells how to use the Interactive Mode emulator, describing each command available.

Section 7 tells how to use the Program Mode emulator, describing the expanded command set and variables available for batch entry.

Appendix describes the interface to and operation of the SCSI device driver. The field of the Task Control Block (TCB) and each direct driver command is defined and described.

1.4 Notation

This manual uses certain terms and notations you should become familiar with. A few examples of these terms follow.

- The terms PED-4010, emulator, and program refer to the Initiator Emulation Program.

- PED-4001 and board refer to the Data Acquisition and Emulation module.

- Generally, acronyms are used in place of program names and frequently used concepts. For instance, CDB is used in place of Command Descriptor Block.

- <> angle brackets refer to a required field or expression you must enter. The type of field required is described within the brackets. For example, <expression> tells you an expression is inserted here.

- [] square brackets refer to a key on the keyboard. For instance, [Enter] refers to the Enter key, otherwise called the [CR] or [Return] key.

  This notation can also refer to an optional expression. For example, [expression] tells you an expression can be inserted here as an option.

- **Commands** you must enter are given in boldface.

- Whenever hexadecimal values are used, the alphanumeric value is followed by an uppercase H. The initial zero is understood (for example, 380H stands for 0380H).

1.5 Hardware and Software Requirements

The following hardware and software is required to operate the PED-4010 emulator:

- IBM PC, XT, AT, or compatible host computer with keyboard.
- Minimum of 512 Kbytes RAM.
- Minimum of one 360 Kbyte floppy disk drive.
- Color/graphics or monochrome display adapter.
- Color or monochrome monitor.
- PC/MS-DOS version 2.0 or later.
- PED-4001 module.
- PED-4010 Initiator Emulation program diskette.

1.6 Reviewing PED-4000 Installation Procedures

The section below provides a brief discussion on unpacking, configuring, and installing the module and PED-4010 emulation program.

1.6.1 Unpacking and Checking the System Components

After receiving your PED-4010 software package, make sure the following items are included:

- 5 1/4-inch floppy diskette (PED-4010 Initiator Emulation Program).
- Loose-leaf PED-4010 Initiator Emulation User's Manual.
- Software registration card.
- SCSI Bus In-line Terminator.
- Board jumper block.

1.6.2 Configuring and Installing the Module and Software

For complete instructions on configuring and installing the PED-4001 module, refer to Section 1.3 in the State Analysis Program User's Manual.

1.6.3 Creating the PED System Diskette

The PED-4010 diskette that comes with the system contains all the files needed to operate the PED-4001 module and PED-4010 emulator. However, it does not contain PC/MS-DOS. You may want to copy all the files from the supplied program diskette to a formatted system diskette and use that new system diskette to run the emulation software.

If you plan to use a hard disk drive, you should copy all PED-4000 files over to the hard disk. The procedures for these copy operations are described here briefly.

Copying to a System Diskette

STEP 1
Install the PC/MS-DOS diskette in drive A: and turn on the computer. When the computer has booted, remove the DOS diskette and put a formatted system diskette in drive A: If the new diskette has not been formatted, enter the following command before removing the DOS diskette:

FORMAT /S [Enter]

Follow the instructions on the computer display to format the diskette.

STEP 2
With a formatted system diskette in drive A:, put the PED-4010 Initiator Emulation program diskette in drive B: and copy all the files from drive B: to drive A: by entering the following command:

COPY B:*.* A: [Enter]

You now have a formatted system diskette in drive A: that contains all the PED-4010 software on it. This should be used as your working diskette. Store the original PED-4010 program diskette in a safe place.

Copying to a Hard Disk

STEP 1
Install the DOS diskette in drive A: and turn on the computer. When the computer has booted, direct the prompt to drive C: (if it is not already). You should see this prompt:

```
C:
```

If not, use this command:

```
C: [Enter]
```

STEP 2
Type the following command to create a directory for PED.

```
MD \PED [Enter]
```

A new directory called "PED" now exists on your hard disk.

STEP 3
Remove the DOS diskette from drive A: and insert the PED diskette in drive A:. Type the following commands:

```
CD \PED [Enter]
COPY A: *.*
```

The hard disk drive now has all the PED files you need to conduct Initiator emulations. Remove the original PED system diskette and store it in a safe place.

Included on the disk is a configuration file (CONFIG.PED) and the SCSI driver file (SCSI.SYS). The configuration and SCSI driver file should be in the root directory. If you don't have a CONFIG.SYS file in the root now, copy CONFIG.PED to the root as CONFIG.SYS. If you do have a CONFIG.SYS file, attach the contents of CONFIG.PED to your CONFIG.SYS file.

NOTE

The programs work best if you don't include ANSI.SYS as a device.

1.7 System Start-Up Procedure

Once you have prepared either the system diskette or the hard disk with the PED-4010 files, you can enter the program.

Follow this procedure:

STEP 1
Boot the system from the system diskette you have prepared (drive A:) or from the hard disk (drive C:).

STEP 2
At the DOS prompt, type the following command.

```
PED4010 [Enter]
```

You are now at the main PED-4000 system menu.

If you want to boot directly into the PED menu, use an AUTOEXEC.BAT file. For instructions on creating an AUTOEXEC.BAT file, refer to your DOS User's Manual.

1.8 Reconfiguring the Module for PED-4010

If your system contains a hard disk drive, there can be a conflict between it and the PED-4001 module in interrupt and DMA channel assignments. Default settings for them are:

Parameter	Default Setting
IRQ	2
DMA	Channel 1

If they conflict, you need to change the jumper settings on the module and the device installation line in the CONFIG.SYS file. Add the following parameters to the SCSI driver device statement in CONFIG.SYS:

```
DEVICE = SCSI.SYS vvv a d i
```

where:

vvv is the vector interrupt, default = 124; legal values are 64 and 255.

a is the PED board number, default = 3; legal values are 1, 2, and 3.

d is the DMA channel, default = 1; legal values are 1, 2, 3.

i is the interrupt, default = 2; legal values are 2, 3, 4, and 7.

A value of 0 or a comma (,) preserves the default, allowing you to modify fields without knowing the other default settings.

For example:

```
DEVICE = SCSI.SYS  0 0 2 7
```

or

```
DEVICE = SCSI.SYS  , , 2 7
```

results in an SCSI at vector 123, address 3, DMA channel 2, and interrupt 7. Or, as another example:

```
DEVICE=SCSI.SYS vector 145 board 2 DMA 1 interrupt 3
```

results in an SCSI at vector 145, address 2, DMA channel 2, and interrupt 3. Only decimal numbers are interpreted, allowing you to comment the line with non-numeric characters.

The following tables list the jumper positions for the interrupt and DMA channels available on the module.

Table 1-1. NCR5380 Interrupt Select Jumpers

Jumpers	Interrupt
E3	IRQ7
E4	IRQ4
E5	IRQ3
E6	IRQ2

Table 1-2. NCR5380 DMA Select Jumpers

Jumpers	DMA Channel
E7, E8	3
E9, E10	2
E11,E12	1

NOTE

Install at most one jumper in locations E3 through E6. Install jumpers E7 through E12 in pairs (e.g. E7 and E8, E9 and E10, or E11 and E12).

1.9 Bus Termination

Both ends of the SCSI cable should be terminated properly. If you are installing the PED-4000 as an end device, use the in-line SCSI bus terminator for that purpose.

CAUTION

Observe correct connector and terminator orientation. Orienting the connector and terminator incorrectly can cause accidental grounding and faulty connection of terminator power which can damage the PED-4000, cable, and other attached devices.

1.10 Terminator Power

The in-line bus terminator uses pin 26 (TERMPWR) to power the internal terminating resistors. TERMPWR must be energized by a single device on the bus. It is recommended that you use the initiator as the power device.

If you want to use the PED-4000 for this purpose, you must install a jumper at E13 on the PED-4001 module. Refer to Section 1.3 of the PED-4001 State Analysis Program User's Manual for information on installing a jumper at this position.

SECTION 2
OPERATIONS AND PROCEDURES

The PED-4010 Initiator Emulation program provides a total SCSI initiator emulation environment integrated with SCSI bus analysis. The program gives you the tools to create a device description of any SCSI target and use this description, referred to as a DDL, to interact with the target device over the SCSI bus.

Figure 2-1 provides a schematic representation of how the various elements of the Initiator Program work together.

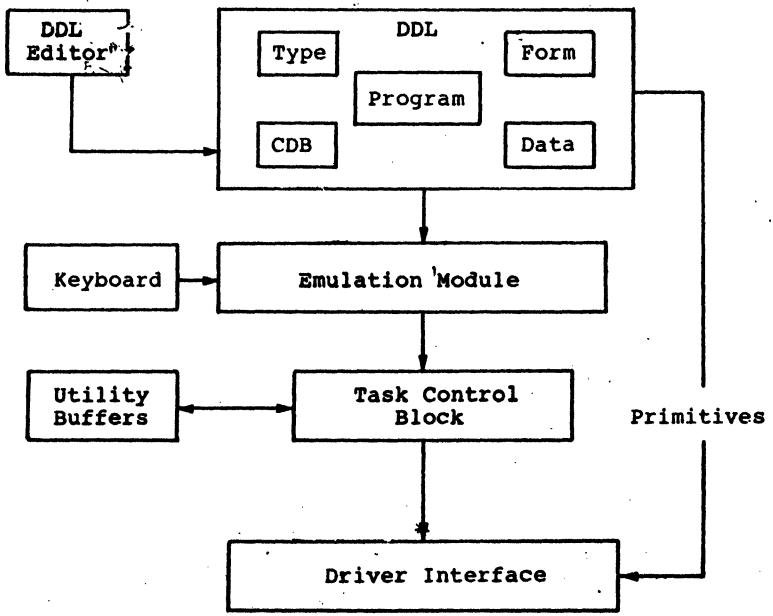


Figure 2-1. Functional Units

2.1 Device Description Library

The Device Description Library (DDL) is resident in RAM and is both created and managed by the Initiator Emulation program. A DDL defines each command and expected data response in the command set of a particular target device. Included in the library is the Command Descriptor Block (CDB) and data buffer (DATA) associated with each command. The DDL also includes templates used in defining CDBs and DATA buffers. A command CDB template is referred to as a command TYPE while a DATA buffer template is referred to as a data FORM.

While many of the device descriptions you will use to analyze the target exist in the DDL, you may choose to create a device description of your own using PED-4010. Whether you use the DDL or create your own definitions, however, the device description consists of five elements: CDB types, CDB definitions, data buffer forms, data buffer definitions, and initiator programs.

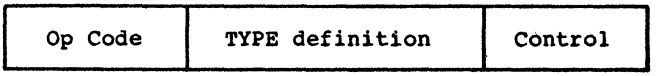
Several DDLs may be resident at one time, allowing for a mix of targets on the SCSI bus. The current target ID assignment is used by the program to automatically select the DDL from the system configuration table.

Each of the elements found in the DDL are described here briefly. For more detailed information on creating and editing DDLs, refer to Section 4.1.

2.1.1 TYPE Definitions

TYPE definitions provide a template to construct CDBs. A CDB can be six, ten, or twelve bytes in length with the first byte always the command Op Code and the last byte always a control byte. The use of the intervening bytes varies according to commands. The CDB TYPE template enables you to identify the command type by name as well as define the nature of all fields between the Op Code and control byte barriers.

CDB TYPE



Each field can be up to 32 contiguous bits in length. Two kinds of CDB fields are identified: Named and Enumerated. Named fields are fields identified by a singular name; all fields larger than eight bits must be named. Enumerated fields are fields with several assigned names, each name conditional on a unique value; enumerated fields cannot be larger than eight bits long.

During an emulation, you can refer to each CDB field by its assigned name. You do not need to define a separate TYPE template for each CDB; several CDBs can share the same TYPE template.

For more on CDB TYPE definitions, refer to Section 4.2.

2.1.2 Command Descriptor Block (CDB)

A CDB is the definition of a particular target command. Each CDB in a DDL is assigned a unique name. During emulation, you can invoke the command by referring to its name as defined by the CDB. CDBs carry default field values. You can modify these field values when you invoke the command by assigning a modified value to the field. You compose a command CDB by using a command TYPE template.

For example, if there were a command CDB called READ with defined fields ADDRESS and LENGTH, then executing the command line

```
READ ADDRESS = 0  LENGTH = 080
```

results in the program setting the ADDRESS field value to 0 and the LENGTH field value to 80 hex.

NOTE

> Defined values for such fields as ADDRESS and LENGTH do not affect the default value of these fields.

A typical CDB for a six-byte command might look like this:

BIT	7	6	5	4	3	2	1	0
0	Operation Code							
1	Logical Unit No			Logical Block Address				
2	Logical Block Address (if required)							
3	Logical Block Address (if required)							
4	Transfer Length (if required)							
5	Control Byte							

Figure 2-3. Sample CDB.

You can tailor a CDB to your own requirements, manipulating addresses and defining the structure of the command to test any specific target.

For more information on using and setting up CDBs, refer to Section 4.2.

2.1.3 FORM Definition

Use the FORM template to define the form of the DDL data buffers much as you use the TYPE template to define the form of a command CDB. DDL data buffers act as a source of data during data output operations, such as WRITE or MODE SELECT commands, or as buffers for data during data input operations, such as INQUIRY or MODE SENSE commands.

A data FORM can be from 1 to 1024 bytes in length, subdivided into any number of fields, each field identified by a unique name. Three kinds of data fields exist: named, enumerated, and text.

For example, if there were a data buffer named SENSED and the FORM template defined fields called ERROR_CODE and SENSE_KEY, then executing the command lines

```
SENSED.ERROR_CODE = 070
SENSED.SENSE_KEY  = 5
```

would result in the ERROR_CODE field being set to 070H and the SENSE_KEY field to 005H.

NOTE

The extension operator (.) is used to separate data buffer names from FORM field names.

For more information on using and setting up FORM templates, refer to Section 4.3.

2.1.4 Running the Initiator Program

The PED-4010 emulation runs programs resident within the DDL or from disk to exercise the SCSI target. Information gathered from such programs can be displayed and saved for later analysis. These programs are written in a programming language which is like structured BASIC but without line numbers. PED-4010 supports both scalar and array variables within this language structure and four data types are allowed: 32-bit long, 16-bit integer, 8-bit character, and 16-byte text. All variables within

a program must be declared before they are referenced.

During an emulation session, data buffers defined in the DDL can be treated as additional variables. There are also a number of system variables which the program can access to facilitate emulation, including .TARGET, .INITIATOR, and .LUN.

For a discussion of program conventions and syntax, refer to Sections 6 and 7.

2.2 System Data Buffers

In addition to the data buffers defined in a DDL, two general-purpose utility buffers are available in host RAM: a 512-byte SMALL buffer and a 64K-byte LARGE buffer. Both are implemented as circular buffers, in the same manner as all other DDL data buffers.

2.3 Run-Time Interpreter

The pivotal function of the initiator emulation is the run-time interpreter. This interpreter scans an input expression, checks syntax, searches command tables, interprets and, if valid, executes the expression. The expression can be from the keyboard, from a DDL program, or from a program file on disk. Syntax and lexical errors are reported to the user.

While this function is transparent to the user, the result manifests itself in two distinct ways. If you have created a program file, the interpreter will go through the file, executing each valid expression as it is encountered. Results of the program execution are either displayed or stored at a selected site. If you choose the interactive mode, each command you enter is immediately processed and the results are displayed.

For more information on interactive mode, see Section 6.

2.4 Configuration Module

PED-4010 includes an SCSI system configuration table which identifies the resident DDL assigned to each SCSI bus address by ID number. The module also assigns an ID number for the currently active target, initiator, and LUN. In this way, you may assign more than one target for emulation, or more than one program for each target selected.

For more information on configuring your system, see Section 5.

2.5 DDL Editor

The Device Description Libraries are managed by the DDL editor. Using this utility, you can create, view, or edit DDLs, TYPEs, FORMs, data, CDBs, or DDL programs.

For a discussion of how to edit the DDL(s), refer to Section 4.

2.6 SCSI Device Driver

All communication between the PED-4001 module and the host system boards is handled by the SCSI Device Driver. It is this program which conveys the PED-4010 emulation software between the two hardware points and drives it. The interface between the device driver and the emulation program is the Task Control Block (TCB) -- a 79-byte block of parameters residing in host RAM.

All bus transactions are initiated by loading of the correct parameter values into the TCB and the generation of a software interrupt (all calls to the driver are through hardware and software interrupts). All interface instructions at the TCB and driver are controlled by the PED-4010 and is transparent to the user.

2.7 Program Utilities

The emulation and analysis program share FILE and HELP programs. The FILE utility allows you to move DDLs to and from disk and to rename and delete existing DDL disk files. The HELP utility provides you with an on-line assistance to every area of the PED-4010 program. The [F1] key gives you immediate, context-selected, help messages.

2.8 Typical Program Paths

During the course of a normal session, you must perform a number of activities. The process of creating an initiator emulation consists of several steps.

1. Creating a File.
2. Defining command types.
3. Specifying CDBs.
4. Defining data buffer forms.
5. Specifying data buffers (if needed).
6. Assigning the device description library to a specific device ID.
7. Creating a batch program, or running the interactive program.
8. Storing and analyzing the results.

All PED-4000 programs support a uniform set of keyboard and display conventions. Learn to use these keys and screen functions.

3.1 Keyboard Functions

There are three kinds of keyboard functions you use during PED-4010 operation:

a. Menu and Data Cursor Movement
b. Function Keys
c. Action Keys

Each is explained here.

Menu and Data Cursor Movement

[← →]	Moves the cursor to the left or right one character each time they are pressed without erasing the previous character. When choosing a file from the device description directory, you must use the up-arrow and down-arrow keys to move from one column of the directory to another.
↑ ↓	Moves the cursor up or down one line each time they are pressed without erasing the character. Use these keys to move the cursor between the DDL directory entries.
[Back Space]	Deletes one character at a time to the left of the cursor, moving the cursor and characters to the left one position.
[Esc]	Causes you to exit the level of the program you are presently in and return to the next higher level.

Below most data or menu windows you will see a quick summary of the possible cursor movements allowed. See more about this under Screen Functions.

Function Keys

There are ten function keys on the PC programmed as hot keys. Pressing any of these causes the following reactions:

[F1]	HELP. Describes the situation and your alternatives at the point where you invoked it.
[F2]	FILE. Enters the emulation file menu.
[F3]	EDIT. Enters the emulation edit menu.
[F4]	CONFIGURE. Enters the emulation configuration menu.
[F5]	RUN. Enters the emulation run menu.
[F6]	SWAP. Shifts between emulation and analysis programs.
[F7]	SETUP. Enters analysis setup.
[F8]	EXIT. Exits to the next higher menu.
[F9]	CAPTURE. Enters the analysis capture routine.
[F10]	DISPLAY. Enters the analysis display routine.

3.2 Display Conventions

The PED-4000 program presents you with a display that consists of windows and entry areas. In general, the display looks like Figure 3-1:

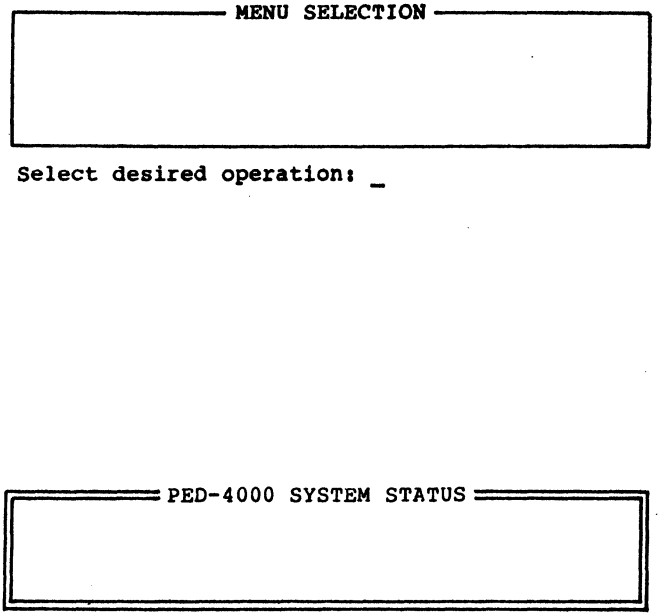


Figure 3-1. Screen Display.

These three areas can be defined as follows:

Data/Menu Window

In this field appear all the menu selections and file options. Choose one of these options either by using the up and down keys or by entering the number of the selection. For instance, if you see these selections

1. EDIT
2. CONFIGURE
3. RUN

press "3" to highlight the RUN option or use the down key. Press [Enter] to select the option and enter the submenu.

Selection windows are identified by their single-line borders.

Entry Area — In this field you enter data and make selections. A prompt and cursor are usually present, as in this example:

```
Select desired operation:_
```

In most situations, you can also make action selections within a program. All response entries must be followed by [Enter]; this causes the selected action to occur.

Status Window — Status windows are identified by their double-line borders. User information is displayed in this window. Reverse, dark, and bright video is used within the windows to enhance recognition. Information displayed within these windows includes PED-4000 system status and SCSI bus status.

Certain screens within the program differ from this standard. These exceptions are explained during the discussions on the respective routines.

3.3 How to Enter the PED-4010 Initiator Emulation

Before you can use the emulator, you must first enter the system. To do this, follow these instructions:

STEP 1
Put the PED-4010 system diskette in drive A. Boot the system. This prompt is displayed:

```
Installing PEDRIVER version 2.10, 8/27/87
```

STEP 2
Bring up the DOS prompt. When it appears, enter

```
PED4010 [Enter]
```

You briefly see copyright information and then the following screen appears:

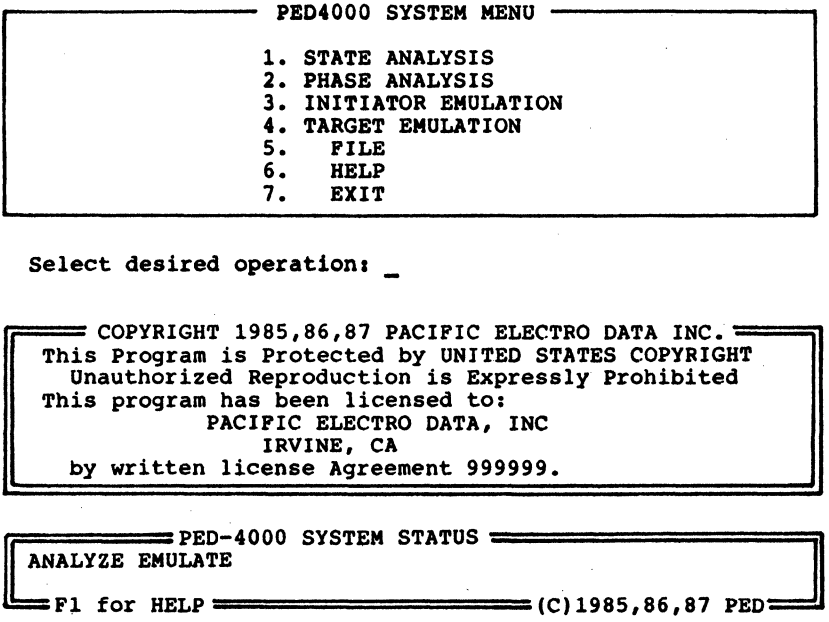


Figure 3-2. Main Menu.

This is the main menu. From here you can enter any of the PED-4000 emulations or analyses installed on your system.

STEP 3
To select the PED-4010 Initiator Emulation program, use [Down] to highlight option 3, "Initiator Emulation," or enter the number "3" at the "Select desired operation" prompt. Press [Enter].

The following window and prompt appears.

```
┌──────────────── INITIATOR EMULATION ────────────────┐
│                                                     │
│                    1. EDIT                          │
│                    2. CONFIGURE                     │
│                    3. RUN                           │
│                    4. FILE                          │
│                    5. HELP                          │
│                    6. EXIT                          │
└─────────────────────────────────────────────────────┘

  Select desired operation: _
```

Figure 3-3. Initiator Emulation Window.

This is the main menu for the PED-4010 Initiator Emulation program. From here you can choose any of the three routines and two utilities you need to create and run the emulation.

Notice that the words "EMULATE" and "INITIATOR" are highlighted in the system status window at the bottom of the display. This tells you that you are in the initiator emulation mode. From this point, you can either enter a routine selection number at the desired operation prompt or use the up and down keys to highlight and select a routine.

Since there are several options to choose from, the next section reviews possible menu paths you can select.

3.4 Menu Path

The Initiator emulation is arranged in a series of routines. Each routine contains a number of menus and submenus you can use to complete the routine task. To get from one place to another in this menu structure, see Table 3-1 below.

Table 3-1. Menu Structure.

1. Edit
   - Emulation Editor
     - Command Type
       - Existing Command Types
       - Command Type Edit
     - Command
       - Existing Command CDB
       - Command CDB Edit
     - Data Form
       - Existing Data Form
       - Data Form Edit
     - Data
       - Existing Data
       - Data Edit
     - Program
       - Existing Run Programs
       - Program Edit
2. Configure
   - Device Assignment
   - SCSI Addresses
   - Driver Options
3. Run
4. Files Utility
5. Help

If you are creating or editing a CDB, data buffer, or program file, you would follow the menu path shown in Figure 3-4:

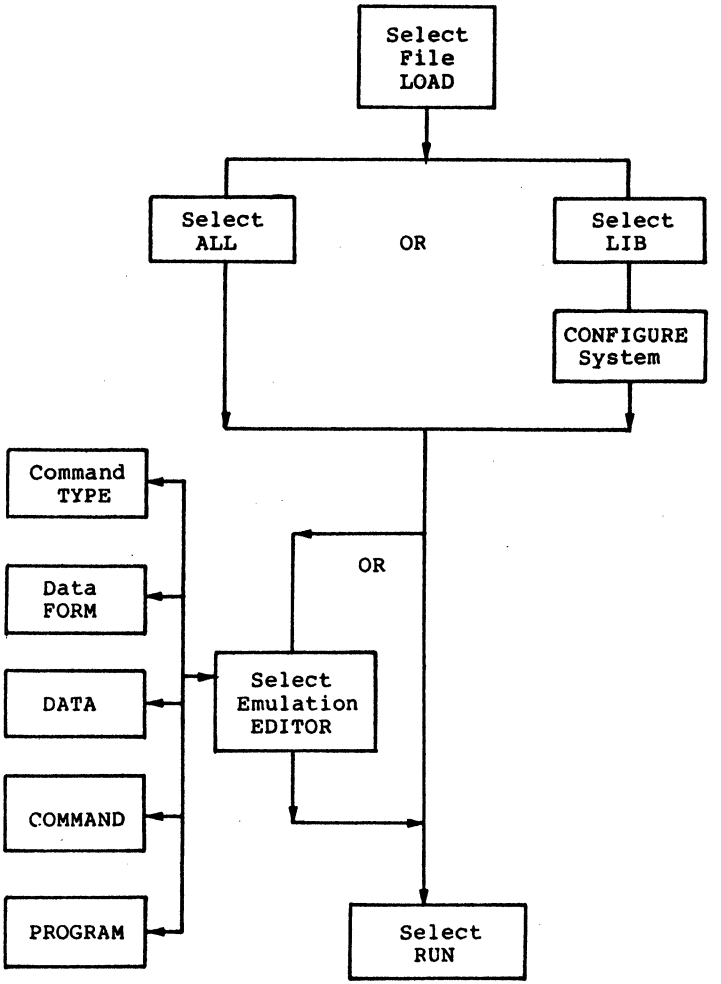


Figure 3-4. Normal Menu Path.

This is one approach to creating/editing a Device Description Library, creating a program, and running the emulator.

In the next section, the procedures for creating or editing a DDL are discussed.

Before you can run the initiator emulation on a selected target device, the emulator must first know what parameters constitute this target: specifically, what are the size, block assignments, and SCSI bus addresses for commands and data buffers recognized by the target device. The emulator enables you to define these parameters using the emulator editor.

Parameters are then stored in the Device Description Library (DDL) for assignment when you run the emulator itself.

4.1 Defining the Device Description Library (DDL)

Before you can begin to define command and data buffers for the emulation, you must first create a DDL.

To create a DDL:

Follow these steps:

STEP 1
Select EDIT from the main initiator emulation menu. The following window and prompt appears:

```
┌──────── Resident Device Description Library(s) ────────┐
│                                                        │
└────────────────────────────────────────────────────────┘
  Select desired operation [Create Alter Rename Delete Exit]:_
```

Figure 4-1. Creating a DDL

If no names appear in the top window, it means there are no DDLs loaded in the computer RAM (random-access memory).

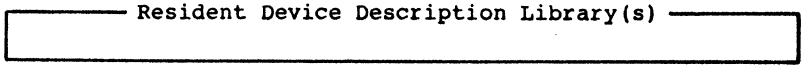
NOTE

If you are accessing libraries or files from computer memory, the word "Resident" appears above many windows. If you are accessing libraries or files from disk, the word "Existing" appears instead.

STEP 2
Highlight CREATE either by using the [Left] and [Right] keys or entering the first letter of the option -- in this case, "C".

Press [Enter].

The following screen appears:



Enter name of desired Library:

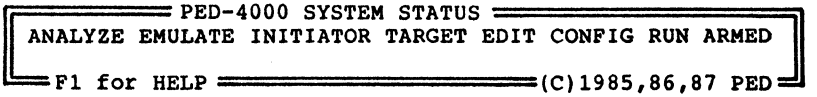


Figure 4-2. Entering DDL Name.

The top window is highlighted and a cursor appears to the right of the prompt "Enter name of desired Library."

STEP 3
Enter a library name. A library name cannot consist of more than thirty alphanumeric characters without extenders.

Press [Enter]. The new library name appears in the top window. You are returned to the previous menu (see Figure 4-1).

If you wish to change the name you have created, select the **Rename** option and enter a new name at the prompt. If you want to delete the name you have just created, select the Delete option. However, be careful. Once you have attached parameters to this library, using the Delete option automatically erases all the values previously entered.

STEP 4
Select the **Alter** option. Press [Enter]. The screen in Figure 4-2 appears displaying your DDL.

If you do not want to assign values to the DDL at this time, press [Esc]. You are returned to the previous menu.

To assign values to this library, press [Enter]. The Emulator Editor menu appears as shown in Figure 4-3.

```
┌──────────────── Emulator Editor ────────────────┐
│                                                 │
│                  1. COMMAND TYPE                │
│                  2.   COMMAND                   │
│                  3. DATA FORMAT                 │
│                  4.   DATA                      │
│                  5. PROGRAM                     │
│                  6.   EXIT                      │
└─────────────────────────────────────────────────┘

  Select desired operation:_
```

```
╔═══════════ PED-4000 SYSTEM STATUS ═══════════╗
║ ANALYZE EMULATE INITIATOR TARGET EDIT CONFIG RUN ARMED ║
╚═══ F1 for HELP ═══ (C)1985,86,87 PED ═══╝
```

Figure 4-3. Emulator Editor

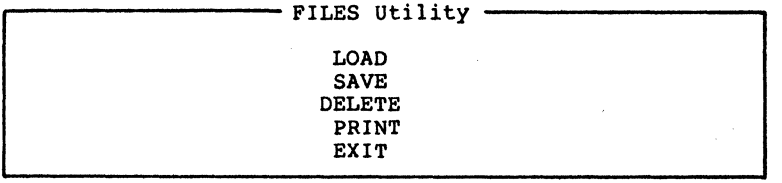
From here you can create or edit the Command Descriptor Block (CDB), the data buffer, or write a program. For more information on creating or editing the CDB or data buffer, see Sections 4.1.1 and 4.1.2.

To edit an existing DDL:

If the DDL already exists as a disk file, use the following procedure to enter and edit.

STEP 1
From the main initiator emulation menu, select the **FILE** option. Press [Enter].

The following window and prompt is displayed:



Select desired operation:_

Figure 4-4. Files Utility Window.

STEP 2
Highlight LOAD by entering "L" at the prompt or by using the cursor keys.

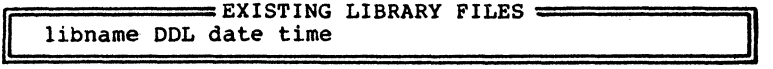
Press [Enter].

The following prompt appears:

Select type of file to load [LIB ALL SETUP DATA]:

STEP 3
Select LIB. Press [Enter].

The following window and prompt is displayed:



Enter name of file to be loaded:

Figure 4-5. Loading DDL Files.

Each file previously created is designated here by its file name, and the date and time it was created. Only a certain number of files can be displayed in the top window at the same time. Use the [Up] and [Down] keys to move from file to file. Use the [Left] and [Right] keys to move the character cursor on the entry prompt line.

STEP 4
Either enter the name of the file you want at the entry prompt, or select it with [Up] and [Down]. Press [Enter].

The DDL file is loaded into memory and the main Initiator Emulation menu is re-displayed.

STEP 5
Once the DDL files are loaded, select **EDIT**.

The window and prompt shown in Figure 4-6 is displayed.

```
┌──────── Resident Device Description Library(s) ────────┐
│Filenm1                          Filenm3                │
│Filenm2                                                 │
└────────────────────────────────────────────────────────┘

  Enter desired operation [Create Alter Rename Delete Exit]:
```

Figure 4-6. Editing DDL Files.

In the top window appears the names of all the libraries you have loaded.

STEP 6
To edit or continue defining CDBs and data buffers, select the **Alter** option.

The screen shown in Figure 4-2 appears.

STEP 7
Enter the name of the library you want to edit or highlight it. Press [Enter].

The Emulation Editor menu appears. You are now ready to edit command and data buffer blocks. You can either edit existing definitions or create new blocks for the DDL files.

4.2 Defining the Command Descriptor Block (CDB)

Before you can exercise the target with the Initiation Emulator, you must first load the Device Description Library with the commands you will use to test the target.

NOTE

> For definitions of the commands a target responds to together with their assigned values, refer to the SCSI interface manual or operations guide for the individual target device.

To define a CDB, complete these two tasks:

1. Defining the Command Type, and
2. Assigning a Command.

This means creating the form of command block you require, then assigning a command word, op code, and default values to that block.

To begin this process, start at the Emulator Editor screen. To get there, refer to Section 4.1. The top window of the screen looks like the following:

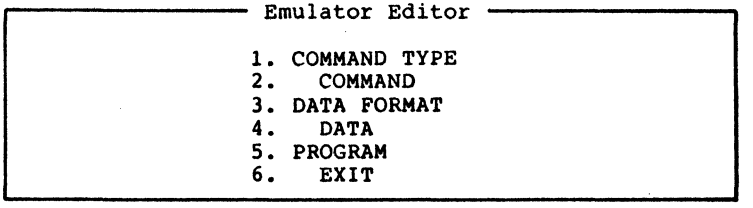


Figure 4-7. Creating the CDB

Each task is described on the following pages.

4.2.1 Creating the Command Format Type

Follow this procedure to create or edit a command Type.

STEP 1
Select COMMAND TYPE from the menu in Figure 4-7. Press [Enter]. The following window and prompt appears:

```
┌──────────── Resident command TYPEs ────────────┐
│                                                │
│  TYPE1                                         │
│  TYPE2                                         │
│  TYPE3                                         │
└────────────────────────────────────────────────┘
 Enter desired operation [Create Alter Rename Delete Exit]:_
```

Figure 4-8. Command Type Selection

STEP 2
Select Create. Press [Enter].

At the prompt

Create name of desired Type:

enter a name. The name must not be longer than thirty characters. No extensions are allowed. Press [Enter].

The following top window appears:

```
┌──────────── Command TYPE EDIT ─────────────────┐
│ BYTE -- bit   7-6-5-4-3-2-1-0                  │
│ 00            Operation  code  Type name: TYPEx│
│ 01            Unit# 0 0 0 0 0  CDB  size: 6 10 12
│ 02            0 0 0 0 0 0 0 0                  │
│ 03            0 0 0 0 0 0 0 0                  │
│ 04            0 0 0 0 0 0 0 0                  │
│ 05            0 0 0 0 0 0 F+L                  │
└────────────────────────────────────────────────┘
```

Figure 4-9. Command Type Edit Window.

Notice that the first byte of the block is reserved for the Op Code and that the first three bits of the second byte are reserved for the Logical Unit Number (LUN). Assign the Op Code when you create the command.

You cannot modify the last six bits of the last byte (in the case of Figure 4-9, bits 5 through 0 of byte 05) from the edit program. These are reserved bits. The last two bits of the last byte (designated in this manual as "F+L") are reserved for the Flag and Link bits.

STEP 3
Choose from one of three command block sizes: six-byte, ten-byte, and twelve-byte. As you move the cursor, notice that the block size increases. When you have selected an appropriate block size, press [Enter].

The cursor jumps to the block.

STEP 4
Using the cursor arrow keys, move the cursor through the block to the beginning bit position. This is the start of a designed field.

Press [Ins]. The bit under the cursor is highlighted. Move the cursor and more bits are highlighted. When you have defined the field you need, press [Ins] again.

STEP 5
If you define a field of eight bits or less, the following prompt appears in the top window:

```
Field type: Named Enumerated
```

Specify whether the field is named or enumerated. Press [Enter].

NOTE

If you define a field of more than eight bits, the program assumes the field is named.

If you specify a named field, then below the field type designation the following prompt appears:

```
Field name =
```

If you specify an enumerated field, the program displays a list of enumerated values in the window similar to the following:

```
Field value: 00 =
Field value: 01 =
Field value: 02 =
```

Up to 256 (0-0FFH) field values may be listed depending on the field size (a one-bit field generates only two field values; a two-bit field generates four values, and so on).

Define a field as enumerated to specify that several values can be associated with the same field.

NOTE

If you want to go back to the block and redefine the field, press [Enter]. The cursor returns to the block. Redefine your field using [Ins].

STEP 6

For a named field: enter a field name (such as LUN, LBA, and so on). You must choose a name unique to the CDB for each named field. Press [Enter] to confirm it.

For an enumerated field: select from the list of enumerated values appearing in the window with the [Up] and [Down] arrow keys. Enter a unique name for the enumerated field. Select the next field value with the [Up] and [Down] arrow keys and name that. When you have name all enumerated values in the field you require, leave the definition by pressing [Enter].

STEP 7

As fields of bits are defined, the program assigns them letters, starting with "a" and proceeding through the alphabet for each field defined, as in this example:

```
                Command TYPE EDIT

    BYTE -- bit    7-6-5-4-3-2-1-0
    00             Operation  code  Type name: DISK4
    01             Unit# a a a a a  CDB  size: 6
    02             a a a a a a a a
    03             0 0 0 0 b b b b  Field type: Named
    04             b b b 0 0 c c 0  Field name: ADDRESS
    05             0 0 0 0 0 0 F+L
```

Figure 4-10. Command Type Edit Window Example.

Notice that enumerated fields are designated by capital letters, while named fields are marked with small letters. Also notice that several fields have not been defined in Figure 4-10. This indicates the block area is reserved.

STEP 8
Once you have defined every field required within the block, press [Esc]. You are returned to the command type directory menu shown in Figure 4-8.

4.2.2 Editing the Command Type

If you want to edit existing command types, follow this procedure.

NOTE

> Be aware that if you edit any fields within a command TYPE that is being used to define existing commands, you run the risk of corrupting or overwriting bits in that existing command.

STEP 1
Select COMMAND TYPE from the menu in Figure 4-6. Press [Enter].

STEP 2
Select **Alter** from the list of options. Press [Enter]

STEP 3
Use the [Up] and [Down] arrow keys to highlight existing command types. When you have chosen one to edit, press [Enter].

The command type block is displayed with the first defined field highlighted. The type and name(s) of the field appear to the right of the block.

STEP 4
Change the field name(s) if you require. To highlight the next field down, press [PgDn]. The field type and name(s) are displayed. Change the values if you require.

Continue down the block, going from field to field. To go back up through assigned fields, use [PgUp].

If you want to change the size of a particular field, press [Enter]. The cursor jumps to the block. Use [Ins] to alter the size of the field within the block. Press [Enter] to return to the field type and name values.

STEP 5
Press [Esc] to leave the command type block. Changes are stored. Select EXIT from the list of operations and press [Enter]. You are returned to the Emulator Editor menu.

4.2.3 Defining a Command CDB

Once you have defined the size of a block for your command and assigned fields to it, you are ready to assign a command name and op code to it. Follow this procedure:

STEP 1
Select COMMAND from the Emulation Editor menu. Press [Enter]. The following window and prompt appears:

```
┌──────────── Resident command CDBs ────────────┐
│ COMMAND1                          COMMAND4    │
│ COMMAND2                          COMMAND5    │
│ COMMAND3                                      │
└───────────────────────────────────────────────┘

Enter desired operation [Create Alter Rename Delete Exit]:
```

Figure 4-11. Command CDB Menu.

In this top window appears a list of all the commands you have created so far.

STEP 2
Select **Create** and press [Enter]. The following prompt appears below the upper window:

Enter name of desired CDB:

STEP 3
Enter a command name. The name can be up to thirty-two characters long, consisting of one or more words. Use an underline (_) to indicate a space between words, such as MODE_SENSE_3. Press [Enter].

The following top window appears:

```
                    Command CDB EDIT
BYTE -- bit    7-6-5-4-3-2-1-0
00             Operation  code   CDB name: XXXXXX
01             Unit# 0 0 0 0 0   CDB type: TYPE1
02             0 0 0 0 0 0 0 0
03             0 0 0 0 0 0 0 0
04             0 0 0 0 0 0 0 0
05             0 0 0 0 0 0 F+L
```

Figure 4-12. Command CDB Edit Window.

where

xxxxx = name you have assigned to the command.
type1 = the command type you define for this CDB.

The cursor is located at the CDB type field.

STEP 4
Use the [Left] and [Right] arrow keys to go through the list of existing command types. As each type appears, the block changes to reflect the type definition.

When you have found the command type appropriate for the command, press [Enter].

The following screen is displayed:

```
                    Command CDB EDIT
BYTE -- bit    7-6-5-4-3-2-1-0
00   00        0 0 0 0 0 0 0 0  CDB name: XXXXXX
01   00        0 0 0 0 0 0 0 0  CDB type: TYPE1
02   00        0 0 0 0 0 0 0 0  CDB data: 0000000
03   00        0 0 0 0 0 0 0 0  Field type: Operation Code
04   00        0 0 0 0 0 0 0 0  Field value: 00
05   00        0 0 0 0 0 0 0 0
```

Figure 4-13. CDB Field Value Assignment Window.

Notice that the first field of the block is highlighted and that a new column of hexadecimal values appears to the right of the byte column. Also, there are several new lines to the right of the block assigned to the block by the command type. These lines indicate the following:

Field Type	Meaning
CDB data	The total value of the block in hexadecimal digits. There are two digits per block line.
Field type	Whether the field is named or enumerated.
Field name	The name of the field as assigned by the command type.
Field value	The hexadecimal value of each field. There are two digits per field line.

You cannot manipulate the CDB data line directly. This line changes as you assign field values. The two-digit bit column to the left of the block also changes to reflect changes in the field value.

STEP 5
At the cursor to the right of the field value prompt, enter the two-digit hexadecimal value for each line of the highlighted field. This value should reflect the op code assignment of that field as stipulated in your SCSI device manual. Allowable values range from 00H through FF.

Press [Enter]. The CDB data line, the bit column line, and the block line all change to reflect the change in the op code assignment.

The next block field is highlighted.

STEP 6
Into each name field enter a default hex value, or select the default enumerated value with the [Up] and [Down] arrow keys. Press [Enter].

Once you have assigned default values to all the fields in the block, you can move up or down the block, review or edit values, using [PgUp] and [PgDn].

NOTE

The program does not let you enter restricted fields, such as the LUN field.

STEP 7
When you are finished defining the field, press [Esc].

You are now prompted to select the default data buffer used when this command is executed. A top window like the one shown in Figure 4-14 is displayed:

```
                    Command CDB EDIT
BYTE -- bit    7-6-5-4-3-2-1-0
00   03        0 0 0 0 0 0 1 1  CDB name: REQUEST_SENSE
01   00        0 0 0 0 0 0 0 0  CDB type: DISK2
02   01        0 0 0 0 0 0 0 0  CDB data: 030000001B00
03   01        0 0 0 0 0 0 0 0  Data data: SENSED
04   1B        0 0 0 1 1 0 1 1
05   00        0 0 0 0 0 0 0 0
```

Figure 4-14. Specifying Data Buffer.

At the "data data:" prompt, select and enter the name of the default data buffer you require when this command is executed. You can choose from any of the data buffers defined in the DDL, as well as the 512 byte SMALL or 64K byte LARGE system buffers.

Use one of the following keys to specify one of these existing data buffers:

Key	Result
[Home]	.SMALL (512 byte) data buffer
[End]	.LARGE (64K byte) data buffer
[Left] or [Right]	Available DDL data buffers. See Section 4.3 for instructions on creating data buffers.

Confirm your selection by pressing [Enter].

STEP 8
You are now prompted to specify the direction of data transfer expected when the command is executed. The top window looks like this example:

```
                    Command CDB EDIT
BYTE -- bit    7-6-5-4-3-2-1-0
00   03        0 0 0 0 0 0 1 1  CDB name: REQUEST_SENSE
01   00        0 0 0 0 0 0 0 0  CDB type: DISK2
02   01        0 0 0 0 0 0 0 0  CDB data: 030000001B00
03   01        0 0 0 0 0 0 0 0  Data data: SENSED
04   1B        0 0 0 1 1 0 1 1  Data phase(s): Either Write
05   00        0 0 0 0 0 0 0 0                 Read None
```

Figure 4-15. Entering the Data Phases.

Use the [Left] and [Right] arrow keys to select a value. Enter your choice by pressing [Enter].

The data phases designate the following:

Phase	Definition
Either	Both directions are allowed with the data buffer; either the source or destination for the target data.
Write	Only allow data transfers from the initiator to the target with the selected data buffer the source of the data.
Read	Only allow data transfers to the initiator from the host with the selected data buffer the destination for the data.
None	Do not allow any data to be transferred.

STEP 9
After assigning data buffer locations, press [Esc]. The previous Edit Emulator menu is displayed.

4.3 Creating Data Buffers

As indicated in the previous section, a data buffer can be assigned to a command. Select between one of two default system buffers or create one of your own. Information placed in these buffers can then be used when the command is executed to exercise the target according to parameters designated by the buffer.

Like CDBs, data buffers are created in two stages:

1. Create the buffer form (Data Form), and
2. Specify buffer parameters within that designated area.

You can also edit buffers by using the Alter option. See the following procedure for creating and editing data forms and buffers.

NOTE

Be aware that if you edit any fields within a data FORM that is already being used to define an existing data buffer, you run the risk of corrupting or overwriting bits in that existing buffer.

4.3.1 Creating Data Forms

The first step in developing a data buffer is to create the buffer template (called a data form). Use the following procedure:

STEP 1
Enter the program and proceed to the Emulator Editor menu as you did in Section 4.1. Remember: you must first load or create a DDL before you can enter the editor menu.

At the menu, select the DATA FORM entry. Press [Enter]. The following menu window appears:

```
┌──────────── Resident DATA FORMs ────────────┐
│ FORM1                              FORM4    │
│ FORM2                              FORM5    │
│ FORM3                                       │
└─────────────────────────────────────────────┘
 Enter desired operation [Create Alter Rename Delete Exit]:_
```

Figure 4-16. Data Form Directory Window.

This window contains all the existing data forms previously created.

STEP 2
Select the **Create** option to define a new data form. Press [Enter].

The program displays this prompt beneath the window:

```
Enter name of desired Form:_
```

STEP 3

Use [BkSp] or [Del] to erase the form in the field and enter a new form name. Press [Enter].

The following window appears:

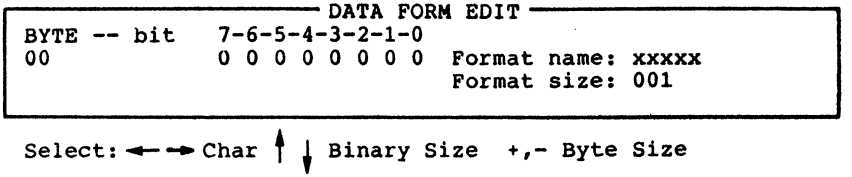


Figure 4-17. Data Form Edit Window.

The cursor is positioned in the Format size field.

STEP 4
Choose the size of the data buffer. Select a size from 1 to 1024 bytes. Press [Up] arrow to increase the format size by 2; press [Down] arrow to decrease by one-half. Press [+] to increase the size by 1; press [-] to decrease the size by 1. Press [Home] to return to zero; press [End] to go to the upper limit of format size. You can also type in a value (less than 400) to specify an exact size. Notice, that as the size increases, the block size increases, and vice versa.

When you have chosen a data buffer size, press [Enter].

The cursor moves to the first bit of the first byte in the data buffer block.

STEP 5
Use the arrow keys to locate the cursor at the beginning bit position of a field you want to define.

Press [Ins] to start the field definition. Use the arrow keys to move to the end bit of the field being defined. The field is highlighted. Press [Ins] again. The field is defined.

STEP 6
If the field you have defined has eight or fewer bits, then you are prompted with this message:

```
Field type: Named Enumerated
```

Use [Right] and [Left] arrow keys to move between these two options. Press [Enter] to select one. If the field you defined is more than eight bits, the field type is designated as Named and the following prompt appears:

```
Field name = _
```

NOTE

A defined field of 8 or fewer bits can be defined as an enumerated type. A named field type can be from 1 to 32 bits in length. A field of from 2 to 16 bytes, starting and ending on byte boundaries, can be defined as a text field type.

If you specify an enumerated field, the program displays a list of enumerated values in the window similar to the following:

```
Field value: 00 =
Field value: 01 =
Field value: 02 =
```

Up to 256 (0-0FFH) field values may be listed depending on the field size (a one-bit field generates only two field values; a two-bit field generates four values, and so on).

Define a field as enumerated to specify that several values can be associated with the same field.

NOTE

If you want to go back to the block and redefine the field, press [Enter]. The cursor returns to the block. Redefine your field using [Ins].

STEP 7
For a named field: enter a field name (such as LUN, LBA, and so on). You must choose a name unique to the data form for each named field. Press [Enter] to confirm it.

For an enumerated field: select from the list of enumerated values appearing in the window with the [Up] and [Down] arrow keys. Enter a unique name for the enumerated field. Select the next field value with the [Up] and [Down] arrow keys and name that. When you have named all enumerated values in the field you require, leave the definition by pressing [Enter].

STEP 8
As fields are defined, the program assigns them letters, starting with "a" and proceeding through the alphabet for each field defined, as in Figure 4-18:

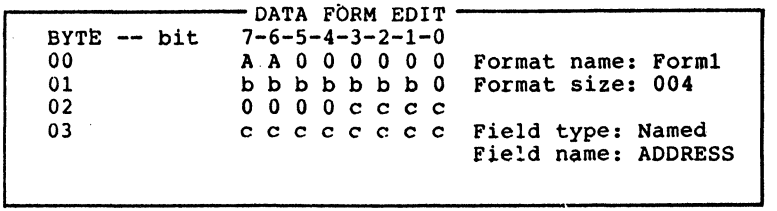


Figure 4-18. Data Form Edit Window Example.

Notice that enumerated fields are designated by capital letters, while named and text fields are marked with small letters. Also notice that several fields have not been defined in Figure 4-18. This indicates the block area is reserved.

STEP 9
Once you have defined every field required within the block, press [Esc]. You are returned to the data forms directory menu shown in Figure 4-16.

The data form you just created appears in the directory window.

4.3.2 Editing the Data Form

If you want to edit existing data forms, follow this procedure.

STEP 1
Select DATA FORM from the emulation editor menu. Press [Enter].

STEP 2
Select **Alter** from the list of options. Press [Enter]. The first data form name is highlighted.

STEP 3
Use the [Up] and [Down] arrow keys to highlight existing data forms. When you have chosen one to edit, press [Enter].

The data form block is displayed with the first defined field highlighted. The type and name(s) of the field appear to the right of the block.

STEP 4
Change the field name(s) if you require. To highlight the next field down, press [PgDn]. The field type and name(s) are displayed. Change the values if you require.

If the highlighted field is enumerated, use the [Up] and [Down] arrow keys to move between the field values. Change these values as required.

Continue down the block, going from field to field. To go back up through assigned fields, use [PgUp].

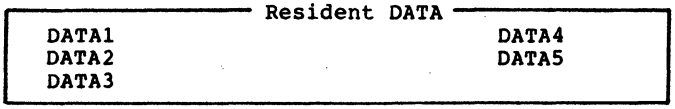
If you want to redefine the size of a particular field, press [Enter]. The cursor jumps to the block. Use [Ins] to redefine the size of the field within the block. Press [Ins] again to return to the field type and name values.

STEP 5
Press [Esc] to leave the data form block. Changes are stored. Select EXIT from the list of operations and press [Enter]. You are returned to the Emulator Editor menu.

4.3.3 Creating a Data Buffer

Use the data format templates to create one or more data buffers for use in writing and running your emulation program. Follow the procedure below for creating a buffer.

STEP 1
Select DATA at the Emulation Editor menu. Press [Enter]. The following window appears.



Enter desired operation [Create Alter Rename Delete Exit]:

Figure 4-19. Resident Data Directory Window.

This window expands as the number of data files grows.

STEP 2
Select **Create** from the option list. Press [Enter]. The following prompt appears:

Enter name of desired Data: _

STEP 3
Move the cursor to the right of the prompt and enter a new data buffer name. Press [Enter]. A menu similar to Figure 4-20 appears:

```
                    DATA EDIT
BYTE -- bit   7-6-5-4-3-2-1-0
00            A A 0 0 0 0 0 0  Format name: Data1
01            b b b b b b b 0  Format size: Form1
02            0 0 0 0 c c c c
03            c c c c c c c c
```

Figure 4-20. Data Edit Window.

The cursor is positioned to the right of the Data format field.

STEP 4
Use the [Left] and [Right] arrow keys to go through the existing data forms. As a data form name appears, the data block changes to reflect that change.

To match a data form to the present data buffer definition, press [Enter].

A window similar to the Figure 4-21 appears:

```
                    DATA EDIT
BYTE -- bit   7-6-5-4-3-2-1-0
00    00      0 0 0 0 0 0 0 0  Format name: DATA1
01    00      0 0 0 0 0 0 0 0  Format size: FORM1
02    00      0 0 0 0 0 0 0 0
03    00      0 0 0 0 0 0 0 0  Field type: Named
                               Field name: ERR_CL
                               Field value: 0
```

Figure 4-21. Entering Field Values.

A new column appears to the left of the data block and several new prompt lines appear to the right.

STEP 5
Enter a field value, if required. The data block and bit column change to reflect the change in the value.

STEP 6
When you have assigned field values to all required fields, press [Esc]. You are returned to the data buffer directory.

4.3.4 Editing the Data Form

If you want to edit existing data buffers, follow this procedure.

STEP 1
Select DATA from the emulation editor menu. Press [Enter].

STEP 2
Select **Alter** from the list of options. Press [Enter]. The first data buffer name is highlighted.

STEP 3
Use the [Up] and [Down] arrow keys to highlight existing data buffers. When you have chosen one to edit, press [Enter].

The data buffer block is displayed with the first defined field highlighted. The type and name(s) of the field appear to the right of the block.

STEP 4
Change the field values, if you require. To highlight the next field down, press [PgDn]. The field type and name is displayed.

Change the values if you require. Press [Enter]. This moves you down to the next field.

If the highlighted field is enumerated, use the [Up] and [Down] arrow keys to move between the field values. Change these values as required.

Continue down the block, going from field to field. To go back up through assigned fields, use [PgUp]. Use [Home] to go back to the top field; use [End] to highlight the last field.

STEP 5
Press [Esc] to leave the data buffer. Changes are stored. Select EXIT from the list of operations and press [Enter]. You are returned to the Emulator Editor menu.

4.4 Deleting and Renaming a DDL Entry

So far you have used the Create and Alter options that appear under the directory windows for Command Type, Command, Data Form, and Data. However, there are two other options you can also use to manage the directory: Rename and Delete.

To Delete a DDL Entry

STEP 1
Use the [Right] arrow key to highlight the **Delete** option. Select Delete from the list by pressing [Enter].

The top entry in the directory window is highlighted and the following prompt appears:

```
Enter name of desired [class]: _
```

where [class] can be TYPE, CDB, FORM, DATA, or PROGRAM.

STEP 2
Use the [Up] and [Down] arrow keys to highlight the entry you want to delete.

When you have highlighted the entry to be deleted, press [Enter]. The entry disappears from the directory window.

To Rename a DDL Entry

STEP 1
Use the [Right] arrow key to highlight the **Rename** option. Select it by pressing [Enter].

The top entry in the directory window is highlighted and the following prompt appears:

```
Enter name of desired [class]: _
```

where [class] can be TYPE, CDB, FORM, DATA, or PROGRAM.

STEP 2
Use the [Up] and [Down] arrow keys to highlight the entry you want to rename.

When you have highlighted the entry to be renamed, enter the new entry name at the prompt. Press [Enter]. The old entry name is replaced by the new one in the directory window.

4.5 Exiting from the Emulation Editor

There are two ways to exit from the editor. Either press [Esc] or select the Exit option from the operation list below the window and press [Enter]. Both return you to the previous menu.

The PROGRAM option in the Editor menu is discussed in Section 5.

4.6 Saving the Results

Your last task is to store the DDLs you have created or edited. To do this, follow this procedure:

STEP 1
Return to the main Initiator menu and select the FILE option.

STEP 2
From the File window, select SAVE. Press [Enter].

STEP 3
The following prompt appears:

```
Select type of file to save [LIB ALL SETUP DATA]:_
```

Select the type of file you want to save. Pick LIB to save a specific DDL file. Pick ALL to save all resident DDLs in RAM to one file. This option also saves all the emulation configuration and State Analyzer setup data in the same file.

If you pick LIB, the following window and prompt appears:

```
┌──────────────── EXISTING LIBRARY FILES ────────────────┐
│ LIB1 DDL [           ]     LIB2 DDL [           ]      │
└─────────────────────────────────────────────────────────┘

Enter name of file to be saved:_
```

Figure 4-22. Resident Library Files Window.

The file name and the date the file was created appears in this window.

If you pick ALL, the following window and prompt appears:

```
┌──────────── EXISTING CONFIGURATION FILES ────────────┐
│ FILE1  ALL [           ]   FILE2 ALL [           ]   │
└──────────────────────────────────────────────────────┘

   Enter name of file to be saved:_
```

Figure 4-23. Resident Configuration Files Window.

The file name and the date the file was created appears in this window.

STEP 4
Enter the name of the file to be saved at the cursor. Press [Enter].

The DDL or configuration file is saved on disk under the selected file name. The first time you save a file, DDL files are given the extension ".DDL"; configuration files are given the extension ".ALL".

4.7 Sample Definition Session

Here is an example of how the Emulation Editor can be used to create a CDB and data buffer.

Take the SCSI command READ USAGE COUNTERS. This command is used by hard disks to track the number of blocks read, the number of seeks requiring carriage motion, the number of correctable or uncorrectable Read errors, and the number of seek errors. Execution of this command sets the usage counters to zero. Counter information is stored in RAM.

This command therefore requires not only the creation of a command block but a data buffer block into which the usage counter information is stored.

The command block as stipulated by the SCSI interface manual is a basic six-byte structure that looks like the following:

BIT BYTE	7	6	5	4	3	2	1	0
0	Operation Code							
1	LUN			Reserved				
2	Reserved							
3	Reserved							
4	Reserved							
5	Reserved						Flag	Link

Figure 4-24. Read Usage Counters Command Structure.

For this example, the op code is specified as 11H.

The data buffer where counter information is stored is also stipulated in the Interface Manual. The structure of this buffer is shown in Figure 4-25.

0	Blocks Read (MSB)
1	Blocks Read
2	Blocks Read (LSB)
3	Seeks (MSB)
4	Seeks
5	Seeks (LSB)
6	Uncorrectable Read Errors
7	Recoverable Read Errors
8	Seek Errors

Figure 4-25. Usage Counter Format

Now that we know the parameters of this command, we can add it to the DDL. The sequence we will use here is:

1. Create Command Type.
2. Create Data Form.
3. Define Data Buffer.
4. Define CDB.

NOTE

Be aware that you don't have to create a new Command Type each time you define a new CDB. Many CDBs are subsets of existing CDBs and therefore use the same Command Type template. Command Type creation is included here to illustrate this specific example.

Follow this procedure.

STEP 1
At the Initiator Emulation main menu, select FILE.

STEP 2
From the FILES Utility, select LOAD.

STEP 3
Select LIB. Choose a specific DDL file from the disk.

STEP 4
Back at the main menu, select EDIT.

STEP 5
Select the Alter option and specify the DDL you just loaded.

This brings you to the Emulation Editor.

STEP 6
Select COMMAND TYPE.

STEP 7
Select Create. At the prompt, enter the new name of the command type. Call it "DISKX."

(Normally you would choose Alter and check the existing command types to see if the specific six-byte block you need for Read Usage Counters already exists before creating a new type.)

STEP 8
In the Command Type Edit window, specify a CDB size of 6. The basic block appears. Since the only field defined for this simple example is the operation code, press [Esc]. You are returned to the main Editor menu.

STEP 9
At the Editor menu, select DATA FORM.

STEP 10
Select Create. Enter the name of the buffer for this command. Call it COUNTS.

STEP 11
At the Format Size prompt, use the [Right] arrow key to expand the size of the data form to nine bytes. Press [Enter].

The cursor jumps to the first bit of the data block.

STEP 12
Look at the buffer depicted in Figure 4-25. The first three bytes are designated Block Read, the second three are seeks, byte 06 is Uncorrectable Read Errors, 07 is Recoverable Errors, and 08 is Seek Errors.

Press [Ins] and use the arrows keys to highlight the first three bytes. Press [Ins] again. This prompt appears:

```
Field type: Named
Field name: _
```

STEP 13
Enter BLOCKS_READ. Press [Enter]. The cursor returns to the data form.

STEP 14
Use [Ins] to define bytes 03 through 05. Press [Enter]. At the field name prompt, enter SEEKS.

STEP 15
Use [Ins] to define byte 06. Give it a field name of READ_ERRORS.

STEP 16
Use [Ins] to define byte 07. Give it a field name of RECOVERED_ERRORS.

STEP 17
Use [Ins] to define byte 08. Give it a field name of SEEK_ERRORS. The data form should now look like the window in Figure 4-26:

```
                    DATA FORM EDIT
    BYTE -- bit   7-6-5-4-3-2-1-0
    00            a a a a a a a a    Format name: Form1
    01            a a a a a a a a    Format size: 004
    02            a a a a a a a a
    03            b b b b b b b b    Field type: Named
    04            b b b b b b b b    Field name: SEEK_ERRORS
    05            b b b b b b b b
    06            c c c c c c c c
    07            d d d d d d d d
    08            e e e e e e e e
```

Figure 4-26. Data Form Edit Sample.

STEP 18
From the Editor menu, select DATA.

STEP 19
Select Create. Enter a new data buffer name. Use the name COUNTERS.

STEP 20
At the data format prompt, use the [Right] arrow key until the format you just created, COUNTS, is displayed. Press [Enter].

The first field, BLOCKS_READ is highlighted and the following prompt appears:

```
Field type: Named
Field name: BLOCKS_READ
Field value: 000000
```

STEP 21
Enter a specified field value for BLOCKS_READ. In this case, 0000A3. Press [Enter].

STEP 22
SEEKS is highlighted. Give this a field value of 0000. Press [Enter].

STEP 23
READ_ERRORS is highlighted. Enter no value (00) for this field. Press [Enter].

STEP 24
RECOVERED_ERRORS is highlighted. Enter no value (00) for this field. Press [Enter].

STEP 25
SEEKS_ERRORS is highlighted. Enter no value (00) for this field. Press [Esc].

STEP 26
Using [Esc], exit DATA and return to the main Editor menu.

STEP 27
At the Editor, select COMMAND.

STEP 28
Select the Create option and specify the command name, in this case call it "Read_Usage_Counters."

STEP 29
At the Command CDB Edit window, use the [Right] arrow key to run through available CDB types until your new Command Type appears. Press [Enter].

The following window appears:

```
                    Command CDB EDIT
  BYTE -- bit   7-6-5-4-3-2-1-0
  00   00       0 0 0 0 0 0 0 0   CDB name: READ_USAGE_COUNTERS
  01   00       0 0 0 0 0 0 0 0   CDB type: DISKX
  02   00       0 0 0 0 0 0 0 0   CDB data: 000000000000
  03   00       0 0 0 0 0 0 0 0   Field type: Operation Code
  04   00       0 0 0 0 0 0 0 0   Field value: _
  05   00       0 0 0 0 0 0 0 0
```

Figure 4-27. Command CDB Edit Example.

STEP 30
Enter the op code specified in the manual. In this case, "11". The window changes to reflect your entry. Press [Esc]. The following field appears:

Data data: .SMALL (512 byte buffer)

STEP 31
Press the [Right] and [Left] arrow keys until COUNTS appears.

STEP 32
Press [Enter]. The following prompt appears:

Data phase(s): Either Write Read None

STEP 33
Select Read.

STEP 34
Press [Enter]. You are returned to the Resident command CDBs menu.

STEP 35
Press [Esc] or select Exit from the option list. You are returned to the Emulation Editor menu. Press [Esc] again to return to the main Initiator menu. Select FILE.

STEP 36
At the File window, select SAVE. Your new files are saved. Exit the program by a combination of ESC and Exit option selections.

You have completed the command and data buffer assignment.

NOTE

READ_USAGE_COUNTERS is a unique vendor command. Therefore, it isn't necessarily useful to all users. However, it does illustrate how easy it is to create a vendor unique CDB using the PED-4000 system.

Before the emulator can use a DDL, you must assign it to a specific SCSI address. The emulator uses the active target address to select the DDL it will access. In PED-4010 you assign SCSI addresses by using CONFIGURE.

Use the CONFIGURE menu to:

- Assign discrete SCSI addresses to the Initiator, Target, and LUNs.
- Re-select of driver options.

5.1 Device Assignment

After you have completed a DDL, you must attach it to the ID of the target address. You may attach a DDL to each of the eight SCSI target IDs by following this procedure:

STEP 1
Enter the main Initiator menu as described in Section 2. If you have not done so already, enter FILE and load the SCSI device DDLs into the computer RAM.

STEP 2
Return to the main Initiator menu and select the CONFIGURE option from the menu list. Press [Enter].

This window and prompt appears:

```
┌──────────────── Configuration Mode ────────────────┐
│                                                    │
│                  1. DEVICE ASSIGNMENT              │
│                  2. SCSI ADDRESSES                 │
│                  3. DRIVER OPTIONS                 │
│                  4.    EXIT                        │
└────────────────────────────────────────────────────┘

 Select desired operation:_
```

Figure 5-1. Configuration Mode Window.

STEP 3
Select the DEVICE ASSIGNMENT option. Press [Enter].

The windows and prompts shown in Figure 5-2 appear:

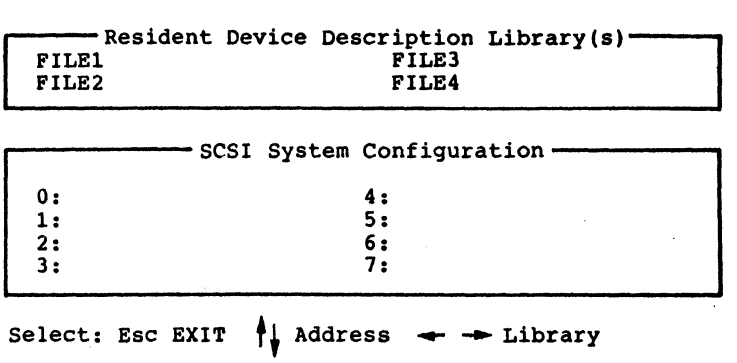


Figure 5-2. Device Assignment Table.

In the top window appear the DDLs resident in the computer. The "0" field in the SCSI system configuration window is highlighted.

STEP 4
Use the [Up] and [Down] arrow keys to highlight the bus ID number of the target.

STEP 5
Use the [Right] arrow key to bring up each existing DDL in turn. The DDL appears to the right of the highlighted ID number. Use the [Left] arrow key to return through the DDL list. The first selection is a blank.

Continue through the Device Assignment table until you have attached DDLs to all the IDs you require.

STEP 6
When you are finished, press [Esc].

Once your targets are properly identified by a DDL, you must select the ID of the active target as well as assign the current LUN and initiator an address.

5.2 SCSI Addresses

Before you can run an emulation, you must attach a DDL to a SCSI ID in the configuration table (defined in Section 5.1) and set the current target, initiator, and LUN IDs. Once set, you may need to reassign these addresses given the following conditions:

- If you have several targets to test, you must reassign the target ID for each emulation you plan to run.
- If you have a target with multiple logical units (LUNs) attached to it, as in the case of target definitions loaded on separate disk drives.
- If you plan to test multiple initiators.

Normally, the Initiator ID will not change. It is assigned a unique ID number. However, you can assign up to eight LUNs for each target, and up to seven different targets.

NOTE

Make sure there is no overlap of target and initiator assignments. For example, if you designate ID 0 as the target address, then the initiator address can't occupy the same ID 0.

To assign SCSI addresses, follow this procedure:

STEP 1
Select the CONFIGURE mode from the main Initiator menu.

STEP 2
Select SCSI ADDRESSES. Press [Enter].

The window and prompt in Figure 5-3 is displayed:

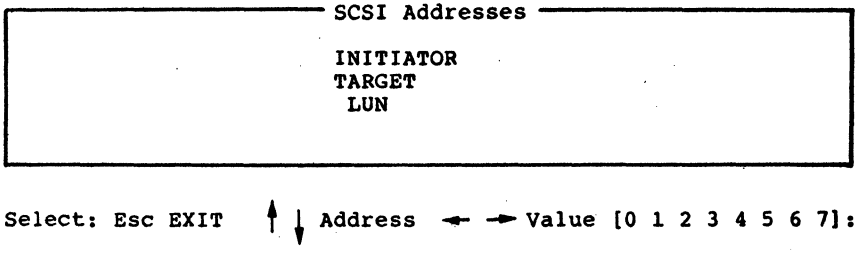


Figure 5-3. SCSI Addresses Window.

STEP 3
Use the [Up] and [Down] arrow keys to highlight one of the three options.

STEP 4
Use the [Left] and [Right] arrow keys to highlight one of the eight possible values.

STEP 5
When you have assigned values to the options you require, press [Esc]. Emulations now use the values assigned.

You can return to this function and reconfigure address values whenever you need to.

NOTE

> You can also assign Target and LUN address values while in either interactive or program mode. For more details, see Sections 6 and 7.

5.3 Driver Options

You can also enable or disable options for your device driver. These options are:

EXTRA	When enabled, the driver will fill the circular buffer and not overwrite it. If more data is sent, it will be thrown away. When more data is requested, zeros will be sent. This avoids buffer overwrite when more data is sent to a buffer than the buffer was designed to hold. Default is NO.
TIME	When enabled, it times out many bus transactions such as READ or WRITE. If bus does not pick up a request within .25 seconds, the driver drops the line. Default is NO.
MESSAGE	Enables or disables message system. This is important in differentiating SASI from SCSI devices. SASI is usually disabled; SCSI is enabled. Default is YES.
DISCONNECT	When the message system is disabled, disconnect is inactivated. When it is enabled, disconnect sets bit 6 of the IDENTIFY message allowing the target to disconnect. Default is YES.
PARITY	Enables parity. Default is YES.
LINK	Enables the link bit. Default is NO.
FLAG	Enables the flag bit. Default is NO.

To change the default driver option, follow this procedure:

STEP 1
Select DRIVER OPTIONS from the Configuration mode menu. Press [Enter].

The window and prompts shown in Figure 5-4 are displayed.

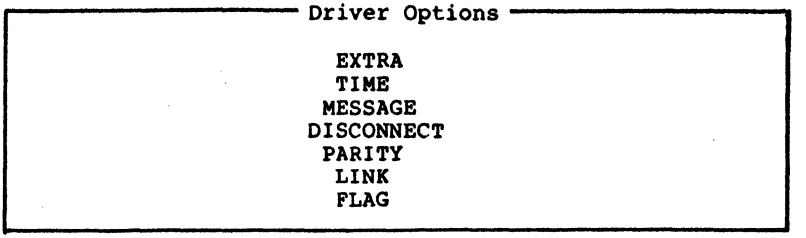


Figure 5-4. Driver Options Window.

STEP 2
Use the [Down] and [Up] arrow keys to highlight an option, or enter the first letter of the option name. The present state of the option appears in the State field below the window.

STEP 3
Use the [Right] and [Left] arrow keys to turn the option ON (Yes) or OFF (No).

STEP 4
When you have configured all the options you require, press [Esc] to assign the driver options and return to the Configuration mode menu.

NOTE

You can also enable or disable many of these options while in the interactive or program mode. For more details on how to do this, see Sections 6 and 7.

SECTION 6
INTERACTIVE MODE EMULATION

There are two ways to run the Initiator Emulation:

- Interactive Mode emulation
- Program Mode emulation

The Interactive Mode is keyboard-driven. In this section, you will learn how to:

- Issue SCSI commands defined in the DDL.
- Assign values to data buffers in the active DDL.
- Issue emulation system commands.

See Section 7 for a discussion of Program Mode.

6.1 Entering Interactive Mode Emulation

Before you invoke the Interactive Mode, create the appropriate DDL and define all command and data buffers required to run the Initiator emulation on the target device.

Follow this procedure to enter Interactive Mode.

STEP 1
At the main Initiator Emulation menu, select RUN. The screen in Figure 6-1 appears:

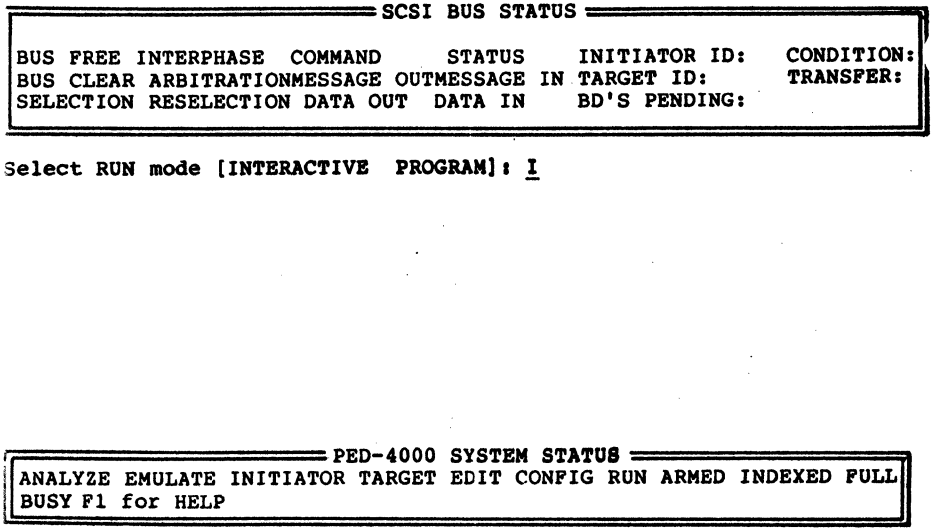
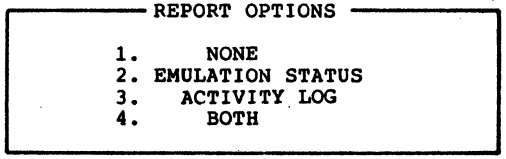


Figure 6-1. Run Mode Screen.

Using this screen you can enter either the Interactive or Program mode. Default is Interactive.

STEP 2
Press [Enter]. Below the Bus Status window, another window appears:



Select REPORT level:_

Figure 6-2. Report Options Window.

When you run the emulator, you can receive several reports on activity. Select NONE to see normal status reports. Select EMULATION STATUS or ACTIVITY LOG to see optional report screens. Select BOTH, to review both the EMULATION STATUS and ACTIVITY LOG report screens. BOTH is the default status mode.

The various status windows are defined below:

Bus Status is always supplied by the top window. This window provides you with information on what the SCSI bus is doing at any specific moment. All messages in this window are basic SCSI bus phase signals. For more information about these signals, refer to your SCSI interface manual or the State Analysis Program User's Manual.

Normal Status is supplied by the middle window if you select the NONE option. This leaves the middle of the screen free for any data you may wish to see during Interactive Mode.

Emulation Status reports on how the emulation is progressing indexed by signals. This is optional and is directly below the bus status window, if selected.

Activity Log provides a historical log of activity on the bus. This is optional and is above the system status window, if selected.

STEP 3
Depending on your requirements, highlight one of the four report options using either the [Up] and [Down] arrow keys or by entering the number which precedes the option you want. Press [Enter].

If you picked either option, a new window appears in the middle. If you picked both options, two windows appear in the middle. If you picked none, no window appears.

Whichever option you choose, this prompt appears above the System Status window:

```
Command = _
```

STEP 4
Enter commands and qualifiers as required. To initiate the command, press [Enter]. Responses to commands appear in both the top and middle windows.

See the next section for a discussion of commands you can use.

.2 Command Lines

line can contain only one command together with any associated ualifiers or arguments. Commands are initiated once you press Enter]. Specific conventions apply to the particular command ypes and are discussed in the following sections.

ommands are of two types:

- o DDL Commands
- o Emulation Commands

DL Commands are the defined CDBs. Define these commands using he Emulation Editor as discussed in Chapter 4.

mulation Commands are the set of pre-defined commands recognized y the emulator. Use these commands at any time during the mulation.

NOTE

> While many emulation commands are shared by both the interactive and program modes, there are several specific to each mode. For more information on this, see Section 6.2.2.

.2.1 DDL Command Assignments: CDB Commands

he first entry on a DDL CDB command line must be a DDL CDB name. nvoke any CDB name from the currently active DDL.

ollow the CDB name by any number of expressions assigning arious DDL TYPE field values. Each expression must be separated y a space.

he general form of the expression must conform to this onvention:

<CDB name> [<TYPE field>=constant...]

or example, if the CCS DDL were active, the command line

```
READ ADDRESS = 0200
```

ould set the ADDRESS field of the READ CDB to 200H and execute he command using that CDB. Results of the emulation would then e saved in the previously-assigned data buffer (.SMALL is efault).

You can assign as many variables per CDB name as you can fit on a line up to a maximum of 70 characters per line. For instance, if you had defined a WRITE CDB, you could invoke a command line looking like this:

```
WRITE ADDRESS = 0400  LENGTH = 0020
```

Where the address field is set to 400H and the transfer length is set to 20H. However, if you were to invoke this command,

```
WRITE LENGTH = 0020  LENGTH = 003F
```

the last assignment of the transfer length value would be used, since the emulator would overwrite the first value.

To assign values to an enumerated field, invoke the enumerated field name as defined by the Command TYPE. If you defined a two-bit field within the CDB WRITE_BUFFER as 00 = HEAD_AND_DATA and 02 = DATA, then you could designate DATA by using this argument:

```
WRITE_BUFFER  DATA
```

This would set the enumerated field of the CDB WRITE_BUFFER to 02. To set the other enumeration of this field, use this argument:

```
WRITE_BUFFER  HEADER_AND_DATA.
```

If you have enumerated fewer than the allowable values per field in your Command TYPE definition, you can use the command line to set others. For instance, in the case of the two-bit field within WRITE_BUFFER, there are only two values enumerated for four possible value entries: 00 = HEADER_AND_DATA and 02 = DATA. This means you can enumerate the 01 and 03 value of this field. To do this, override an existing enumerated value and set it to another field value, as in this example:

```
WRITE_BUFFER  DATA = 03
```

In effect this creates an enumerated value using the existing enumerated label of DATA. This does not effect the default value of DATA, merely creates an enumerated value for this one command only.

6.2.2 Emulation Commands

Besides setting command and data fields, you can also use many emulation commands to exercise and analyze the target. These commands are listed below, grouped by categories.

Configuration: commands that assigns SCSI addresses.

```
INITIATOR
LUN
TARGET
```

Driver Control: commands that control the driver.

```
DISCONNECT          MESSAGE
EXTRA               PARITY
FLAG                RESET *
LINK
```

Diagnosis: commands that report the emulator state.

```
ALL                 REPORT
DUMP                VIEW
MARK
```

State Analysis: commands that control the State Analyzer while in emulation.

```
ARM *
DISARM *
INDEX *
```

System: commands that control the emulator.

```
BYE
CLEAR *
EDIT
CONFIGURE
```

* = used in both Interactive and Program mode.

Consult the following pages for discussions of each command. Commands are listed alphabetically.

ALL

Syntax:

ALL [<DDL_name> <Element_class>]

Function:

Use this command to view size and address locations for either the active DDL or another specified DDL. The ALL report looks like this example:

```
DEMO

The active .DDL is DEMO

aaaa  allocated bytes at bbbb
Table cccc  Free dddd
```

Figure 6-3. ALL Data Window.

where:

aaaa = number of bytes allocated for this DDL.
bbbb = address location where DDL begins.
cccc = base address for table of last-referenced DDL.
dddd = address location from where code starts compiling.

Comments:

Use ALL without parameters to show resident DDLs per box.

Use ALL with parameters to show resident entries per class. Both parameters must be included. Allowable parameters are listed below.

Parameters:

DDL_name = name of specific DDL.

Element_class = TYPE, FORM, DATA, or PROG.

ARM

Syntax:

ARM

Function:

Use this command to arm the State Analyzer. When invoked, the message ARMED is highlighted in the System Status window at the bottom of the screen.

Comments:

Use this function if you intend to capture data using the State Analyzer. To disarm the analyzer, use the DISARM command.

BYE

Syntax:

BYE

Function:

Use this command to quit the Interactive Mode and return to the main Initiator Emulation menu.

CLEAR

Syntax:

CLEAR [name] [value]

Function:

Use this command to clear the initiator's data buffers. .OFFSET and .FILE buffers are automatically cleared to zero.

Comments:

Invoking the command without parameters clears the .LARGE buffer to zero.

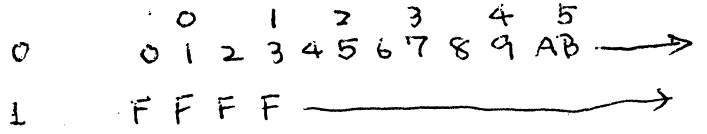
Invoking the command with a buffer name clears the specified buffer to zero.

When invoked with a buffer name and number, the program uses the number to create a pseudo-random sequence in the named buffer. For instance, a value of 2 gives an incremented sequence; 3 provides a decremented sequence. Any value greater than 8 adds the low 8 bits to the high 8 bits. Sequences are repeated every 256 bytes. Default buffer is .LARGE.

Parameters:

name = .LARGE, .SMALL, or a DDL DATA name.

value = 0, 1, 2, 3, and so on.



CONFIGURE

Syntax:

CONFIGURE

Function:

Use this command to enter the CONFIGURE mode while in the Interactive mode and assign a DDL to a SCSI Bus ID.

Comments:

Invoke CONFIGURE by itself to enter the CONFIGURE mode. Use the command to check and edit the current SCSI system configuration.

When you are finished, press [Esc] to return to the Interactive mode.

DISARM

Syntax:

DISARM

Function:

Use this command to disarm the Logic Analyzer. When invoked, the message ARMED is not highlighted in the System Status window at the bottom of the screen.

Comments:

Use this function if you do not intend to capture data using the Logic Analyzer. To arm the analyzer, use the ARM command.

DISCONNECT

Syntax:

DISCONNECT [ON/OFF]

Function:

Use this command to enable or disable disconnects on the current target device. If the disconnect is ON, disconnect is enabled. If disconnect is OFF, disconnect is disabled. Default is ON.

Comments:

Invoke this command without parameters to see the current disconnect state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the disconnect state.

SCSI devices require disconnect to be enabled; SASI devices usually require disconnect is disabled.

This function is also controlled by the CONFIGURE mode. See Chapter 4.

DUMP

Syntax:

DUMP [buffer_name]

Function:

Use this command to view the contents of the specified .SMALL, .LARGE, or user-defined data buffer. The dump display looks like this:

```
Displayed buffer = SAMPLE_BUFFER
 0-1-2-3-4-5-6-7-8-9-A-B-C-D-E-F-...
00000000000000000000000000000000...
00000000000000000000000000000000...
00000000000000000000000000000000...

Command or buffer: _
```

Figure 6-4. Buffer Dump Window.

Comments:

After invoking a DDL command, use this emulation command to review the code dumped into the attached data buffer. Only specify one buffer per command.

If you specify the .LARGE (64K) buffer, you can advance through the buffer by pressing [Enter]. This takes you forward one block at a time. To go back a block, press [-] and [Enter]. To specify a particular buffer block, enter a buffer number at the cursor and press [Enter], or enter another command to exit the dump window.

Go forward in the .LARGE buffer a specified number of blocks by using the command, +n, where n is the number of blocks. Go backward a specified number of blocks by using the command, -n, where n is the number of blocks.

You can display a maximum 512 bytes on the screen.

EDIT

Syntax:

EDIT [<DDL_name> <element_class> <name>]

Function:

Use this command to edit specified DDL formats while still in the interactive mode.

Comments:

Invoking EDIT without parameters causes the program to go through all menu selections. Use the optional parameters to specify DDL_name, element_type, and name. Leave spaces between parameters.

When you are finished, the program returns to Interactive Mode (even if you used the menu mode to select the entry to edit).

Parameters:

DDL_name	Specify name of the particular DDL you want to edit.
element_class	Indicate Command Type, Command, Data Form, or Data options.
name	Specify existing TYPE, CDB, FORM, DATA, or PROG.

For example:

EDIT SG FORM SENSE

tells the program you want to edit the DDL file called SG for a data form called SENSE.

You are taken to that block for editing.

EXTRA

Syntax:

EXTRA [ON/OFF]

Function:

Use this command to enable or disable an extra buffer on the current target device. Default is OFF.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

When enabled, the driver will file the circular buffer and not overwrite it. If more data is sent, it will be thrown away. When more data is requested, zeros will be sent.

This function can also be controlled by using the CONFIGURE mode. See Section 5.

FLAG

Syntax:

FLAG [On/Off]

Function:

Use this function to enable or disable the flag bit on the current target device. Default is OFF.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

INDEX

Syntax:

INDEX

Function:

Use this command to force the logic analyzer to save time and state data around some key index event.

Comments:

The command halts the capture operation after a specified number of writes to the acquisition memory, protecting data saved on either side of the INDEX command. When the analyzer is armed (see ARM and DISARM commands), time and state data is written in memory starting at address 000. At each write, the memory address register is incremented by 1. At a certain point the data will wrap back around on itself and begin to overwrite unless INDEX is invoked.

INDEX controls the capture process, loading the index count value, setting the counter to increment at every write, forcing a write to the memory when time and state data occurs, and halting capture when the index counter reaches its maximum count.

Once you have issued this command, the system status window highlights the word "INDEXED".

INITIATOR

Syntax:

INITIATOR [value]

Function:

Use this command and argument to assign a SCSI address to the current target device ID.

Comments:

Invoke this command without parameters to see the current device address assignment. The program then prompts for a change of address.

Invoke this command with a value from 0 through 7 to change the SCSI device address. A test is performed to determine whether there is any conflict with the target assignment. (If the value is greater than or equal to zero, less than or equal to 7, and is not equal to the target, the new value is used. Otherwise, you will be prompted for a new value.)

This function is also controlled by the CONFIGURE mode. See Chapter 5.

LINK

Syntax:

LINK [YES/NO]

Function:

Use this function to enable or disable the link bit for the current target device. Default is OFF.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

LUN

Syntax:

LUN [value]

Function:

Use this command and argument to assign a LUN address to the current target device ID.

Comments:

Invoke this command without parameters to see the current device address assignment. The program then prompts for a change of address.

Invoke this command with an address number from 0 through 7 to change the SCSI device address, otherwise, it will prompt you.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

MARK

Syntax:

MARK [ON/OFF] [value1] [value2]

Function:

Use this command to mark blocks you write to the .LARGE (64K) buffer.

Comments:

MARK instructions reside in the 6-byte header of each 512-byte block. The first two bytes name the file, the next four name the block in this manner:

When you use this command, each block sent to the buffer is marked with a block number. Block numbering is automatic. File numbering is user-defined.

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state. Use the MARK ON argument without additional parameters invoke the automatic block numbering function.

Follow the command with a value to assign a specific file number.

Follow the command with a second value to assign a specific block number to the block. From here, the designated number is incremented for each new block. If you don't want to change the file number but do want to change the block number, enter a "-1" for the first value, as in this case:

```
MARK -1  100
```

NOTE

The CLEAR command clears both the block and file numbers.

MESSAGE

Syntax:

MESSAGE [ON/OFF]

Function:

Use this function to enable or disable the message system of the current target device. Default is ON.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

This is important in differentiating SASI from SCSI devices. SASI usually disables the message system; SCSI enables it.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

PARITY

Syntax:

PARITY [ON/OFF]

Function:

Use this function to enable or disable the parity bit on the current target device. Default is ON.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

REPORT

Syntax:

REPORT [ON/OFF]

Function:

Use this command to enable or disable the emulator report function.

Comments:

Invoke this command without parameters to see the current state. If it is disabled, the word "OFF" appears above the command prompt until you enter the next command. If the word "ON" appears above the command prompt, it is enabled.

Invoke this command with either ON or OFF to change the state.

RESET

Syntax:

RESET

Function:

Use this command to issue a hardware reset to the bus.

Comments:

Invoke this command when you want to reset the bus to initial states and begin emulating. It also clears all the internal driver assignments.

TARGET

Syntax:

TARGET [value]

Function:

Use this command and argument to assign a SCSI address to the target device.

Comments:

Invoke this command without parameters to see the current device address assignment. The program then prompts for a change of address.

Invoke this command with a value from 0 through 7 to change the SCSI device address. A test is performed to determine whether there is any conflict with the initiator assignment. (If the value is greater than or equal to zero, less than or equal to 7, and is not equal to the target, the new value is used. Otherwise, you will be prompted for a new value.)

This function is also controlled by the CONFIGURE mode. See Chapter 5.

VIEW

Syntax:

VIEW

Function:

Use this command to view all of the parameters defined for the last TCB used. The image of the TCB appears in the middle window. The image looks something like this:

```
     Image of OUR_TCB
TCB_GATE        00   TCB_PRIMITIVE  00         TCB_STATE       0000
TCB_ID_MSG      00   TCB_CDB        00000000000000
TCB_STATUS      00   TCB_COMPLETION 00         TCB_INITIATOR_ID 00
TCB_COUNT_SPC   0000 TCB_COUNT_ACT   0000     TCB_TARGET_ID     00
TCB_DATA_SEG    00   TCB_DATA_OFFSET 0000     TCB_LUN_ID        00
TCB_OPTION      00   TCB_BLK_SZ      0000
TCB_MESSAGE     000000000000000000000
TCB_DMA_MODE    00   TCB_CURRENT_MODE  00     TCB_SAVED_MODE    00
TCB_DMA_HIGH    00   TCB_CURRENT_HIGH  00     TCB_SAVED_HIGH    00
TCB_DMA_ADDR    0000 TCB_CURRENT_ADDR 000    TCB_SAVED_ADDR   000
TCB_DMA_COUNT   0000 TCB_CURRENT_COUNT 00    TCB_SAVED_COUNT  000
TCB_ACT_COUNT   0000 TCB_CURRENT_PHASE 00    TCB_SAVED_PHASE   00
TCB_INTERRUPT_LOCATION 0000     TCB_RETURN_LOCATION   0000
TCB_LINK_SEGMENT 00 TCB_LINK_OFFSET   0000
TCB_ASSIGNMENT 0000 TCB_GROUP         0000
```

Figure 6-5. TCB View Window.

Comments:

Use this command to view the most recent TCB settings for the device driver. Loading another device file into the program causes the TCB image to change. You cannot change the TCB image using this command, nor can you change most of the TCB parameters using the interactive emulation.

See Section 7 and Appendix A for information on modifying the TCB image.

Table 6-1. Interactive Command Summary

Command	Function
ALL	Views size and address locations for current DDL.
ARM	Arms the State Analyzer.
BYE	Quits the Interactive mode.
CLEAR	Clears the initiator's data buffers.
CONFIGURE	Enters the Configure mode while in the Interactive mode and allows ID assignment.
DISARM	Disarms the State Analyzer.
DISCONNECT	Enables or disables the disconnect on the current target device.
DUMP	Views the contents of the specified data buffer.
EDIT	Edits specified DDL formats while in the Interactive mode.
EXTRA	Enables or disables an extra buffer on the current target device.
FLAG	Enables or disables the flag bit on the current target device.
INDEX	Forces the Logic Analyzer to save time and state data around a key index event.
INITIATOR	Assigns a SCSI address to the current target device ID.
LINK	Enables or disables the link bit for the current target device.

Table 6-1. Interactive Command Summary (continued).

Command	Function
LUN	Assigns a LUN address to the current target device ID.
MARK	Marks blocks you write to the .LARGE buffer.
MESSAGE	Enables or disables the message system of the current target device.
PARITY	Enables or disables the parity bit on the current target device.
REPORT	Enables or disables the emulator report function.
RESET	Issues a hardware reset to the bus.
TARGET	Assigns a SCSI address to the current target device.
VIEW	Views all parameters defined for the last TCB used.

The PED-4010 Initiator enables you to create programs for running multiple emulation operations without having to enter commands after the completion of each one. Once the program is activated by the emulator, it runs through all operations stipulated and puts the data into buffers specified by the program.

In this section you will learn how to:

- o Enter Program mode.
- o Write a program.
- o Use operators.
- o Use reserved words and system variables.
- o Invoke the program mode emulator.

7.1 Enter Program Mode

Before you can write a program you must enter the program mode emulator. Follow this procedure to enter the program mode:

STEP 1
Select FILE and load the DDL files you want to use. If the proper files are not loaded, the emulator will not understand your DDL commands.

You are returned to the main Initiator menu.

STEP 2
Select EDIT. This window is displayed:

```
┌──────────── Resident Device Description Library(s) ────────────┐
│ DDL1                                DDL3                       │
│ DDL2                                DDL4                       │
└────────────────────────────────────────────────────────────────┘

  Select desired operation [Create Alter Rename Delete Exit]:_
```

Figure 7-1. DDL Selection Window.

STEP 3
Select the DDL you want to use (Alter), or create a new DDL. (See Chapter 3 for details on creating a DDL.)

The window shown in Figure 7-2 appears.

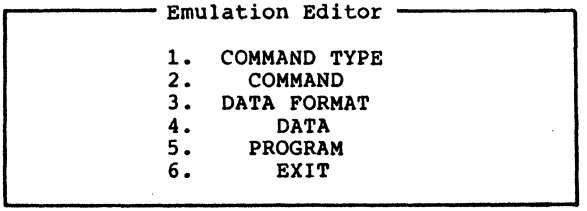


Figure 7-2. Emulation Editor Window.

STEP 4
Select PROGRAM. You are presented with a list of programs already existing in this file, as shown in Figure 7-3.

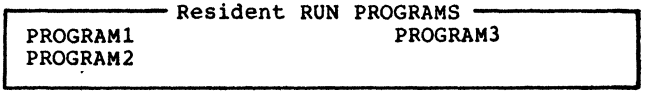


Figure 7-3. Program Selection Window.

STEP 5
If you creating a new program, select the **Create** option. (If you are editing an existing program, select the **Alter** option.)

STEP 6
Select the resident program you want to edit or enter the name of the new program you want to create. Press [Enter].

The window in Figure 7-4 appears:

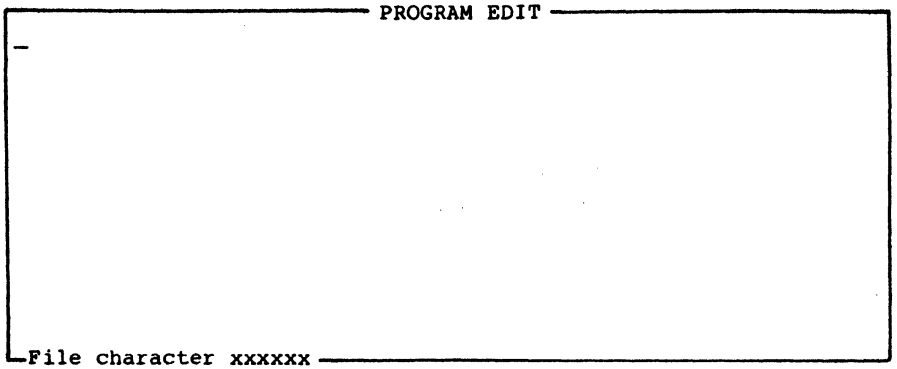


Figure 7-4. Program Edit Window.

7.2 Using the Program Edit Window

There are several function keys to help you edit the program while you are in the Program Edit window. These are defined in the list below:

Keys	Function
Cursor Keys ↑ ↓ <- ->	Move left or right along a line using the [Left] and [Right] arrow keys. Move up or down using the [Up] and [Down] arrow keys.
[PgUp] [PgDn]	Move up or down approximately 20 lines.
[Home]	Move to the top of the program.
[End]	Move to the end of the program.
[Ctrl] Y	Delete the line under which the cursor is positioned.
[Ctrl] T	Delete the word under which the cursor is positioned.
[Ctrl] G or [Del]	Delete the character under which the cursor is positioned.

Keys	Function
[Backspace]	Delete the character to the left of the cursor.
[Esc]	Exit from the program. Return to the Emulation Editor menu.

In the lower left corner of the Program Edit screen is the file character counter. The counter changes each time the cursor moves to a different line of the program. Use this counter to keep track of the number of characters on each line, the number of characters in the program.

Using the function keys, write in your program. Program conventions and commands are simple, resembling a structured BASIC but without the line numbers. Both scalar and array variables are supported and four data types.

Conventions and commands are explained in the next sections.

7.3 Program Conventions

Observe these conventions while writing your program:

1. Variables must be declared first before any instructions. Therefore, the first line of the program must include a statement of the variables to be used in the program. See 7.3.1 for details.

2. Use only one statement per line. This means one command word together with arguments and qualifiers can go on a line. A maximum of 70 characters can fit on a line.

3. Indentation is not required. However, you may find it easier to keep track of statements if you indent them in this way:

```
A ----------------
  B ----------------
    C ----------------
D ----------------
```

4. Line numbers are not supported.

5. Use only valid language and symbols. This includes:

   Special characters (see Section 7.3.2)
   Constants (see Section 7.3.3)
   Operators (see Section 7.3.4)
   Commands (see Section 7.3.5)

6. Precede each comment on a line by a semicolon (;). For other special character conventions, refer to Section 7.3.2.

7. Use either lower- or uppercase alphabetic characters to compose your program. (The program processes uppercase slightly faster than lowercase.)

8. 'Space' is a delimiter and is required between words, symbols, and argument groupings. You may also use a comma as a space delimiter in variable declarations.

For detailed information on commands, statements, and syntax, see the sections below.

7.3.1 Variable Types and Naming Conventions

Conventions you must follow in naming variables are listed below:

1. All variables must begin with alphabetic characters.

2. Variable names can contain either upper- (A-Z) or lowercase (a-z) alphabetic characters, numbers (0-9), and spaces designated by underscores (_). The program converts all alphabetic characters to uppercase internally.

3. No operators are allowed in variable names.

4. Variables used in a program must be declared on the first line(s) of that program.

5. A variable name cannot match a CDB name. If this happens, you will get a syntax error.

Variables are divided into a number of different types. The allowed variable types you can declare include:

Data:

Type	Size
LONG	32 bit. Signed.
INT(EGER)	16 bit. Signed.
CHAR(ACTER)	8 bit. Signed.
TEXT	16 bytes. ASCII.

There can be a singular occurrence of a variable (called a scalar variable), or a multiple occurrence of the same variable (called an array variable).

Scalar variables are the default and include the variable types LONG, INT, CHAR, and TEXT.

Variables are identified as array in statements when they are indexed in the following form:

```
TYPE variable[array size]
```

For example,

```
ndigit[10] = ndigit[0], ndigit[1]... ndigit[9]
```

Arrays are only one-dimensional and the size is a constant (usually a decimal). The array size should be enclosed by square brackets; no space is allowed between the variable and its array size value.

All variables must be assigned types. Arithmetic is done internally as 32 bits and variables are sign-extended to fit.

Use one line for each variable type declared. For instance, if you were declaring the variable "size" as a long and "cursor" as an integer, you would use this format:

```
long size
int cursor
```

As you can see, the type declaration is followed by the word or character being declared as a variable. If you are declaring several variables of the same variable type, you can declare them in the same line, like this:

```
int cursor, repeat a array[10]
```

Reserved Words

You cannot use any of these reserved words as variables:

ARM	BREAK	CASE	CHAR	CONTINUE
DEFAULT	DISARM	DO	ELSE	ENDIF
ENDSTRUCT	ENDSWITCH	FOR	GOTO	HALT
IF	INDEX	INPUT	INT	LONG
NEXT	NIF	PRINT	PROCEDURE	RETURN
STRUCT	SUSPEND	SWITCH	TEXT	UNTIL
WHILE				

7.3.2 Special Characters

Certain special characters have special meaning. The emulator recognizes these characters and treats the word or character with which it is associated in a specific way.

These special characters and their meanings are listed below.

Characters	Meaning
0	When preceding a number, indicates hexadecimal value. For instance, 070 tells the emulator to evaluate this number as hexadecimal.
#	When preceding a symbol, indicates hexadecimal output. For example, #x tells the emulator to print the number "x" in hexadecimal.
;	Preceding a string, indicates the remainder of the line is a comment. For example, ";this is a comment line" tells the emulator this is a comment string.
[Enter]	Indicates the end of the line.
" "	Encloses an ASCII text string. For example, "this is a text string" tells the emulator that the characters inside the quotation marks is an ASCII string.
[ ]	Encloses an array index. For example, CURSOR[10] tells the emulator this statement is array and should be treated as multiple examples of the same variable. No space is allowed between the variable and its index size.

When used as a prefix to a word, indicates the word to follow is a system variable command. For example,

```
.PARITY
```

tells the emulator that this is a system variable command applying to the state of the parity bit in this program. For information on System Variable commands, refer to Section 7.3.5.

7.3.3 Constants

Several characters are defined as constants. These characters maintain their value wherever they are found.

These constants are shown below:

Constants	Description
0 - 9	Standard numbers are used in decimal form.
0 - 9 A - F	Hexadecimal format requires a 0 prefix, as in this example: 0700 The letters A through F are also valid characters for hex format.

7.3.4 Operators and Expressions

Operators are symbols indicating the type of operation the emulator should perform on one or more values (variables or constants). Operators involve logical, mathematical, and relational operations. Expressions are a series of values separated by operators.

Operators

The operators used as conventions in the program mode are described below:

Symbol	Function
+ -	Additive: + = "add" as in 4 + 3 = 7. - = "subtract" as in 6 - 4 = 2.
* / %	Multiplicative: * = "multiply by" as in 3 * 6 = 18. / = "divide by" as in 6 / 3 = 2. % = modulo divide as in 7 % 4 = 3.
< <= == > >= !=	Relational: < = "less than" as in 4 < 5. > = "greater than" as in 5 > 4. =< = "less than or equal to" as in x =< y. => = "greater than or equal to" as in y => z. != = "not equal to" as in 2 != 1. == = "equal to" as in 2 == 2
& | ^ && ||	Logical: & = binary AND. | = binary OR. ^ = binary exclusive OR. && = logical AND. || = logical OR.

= -= += *= /= ~ ^= |= &=	Assignment: A = B --> A = B A -= B --> A = A - B A += B --> A = A + B A *= B --> A = A * B A /= B --> A = A / B A ~= B --> A = 0 - B A |= B --> A = A <logical OR> B A ^= B --> A = A <logical exclusive OR> B A &= B --> A = A <logical AND> B where: A = a named variable. B = an expression.

Expressions

There are two types of expressions allowed by this program:

- o Arithmetic expressions
- o Logical expressions

Arithmetic expressions take this form:

<variable> <assignment operator> <value> [<operator> <value>...]

where,

<assignment operator> can be one of the assignment operators (=, -=, +=, *=, /=, ~=, |=, ^=) or one of three binary logical operators (&, |, ^).

<operator> can be any one of the assignment operators listed above plus one of the additive (+, -) or multiplicative (*,/,%) operators. (Relational operators, &&, and || operators cannot be used.)

<value> can be either a variable or a constant.

Therefore, the first operator must be an assignment operator. Subsequent operators may include arithmetic operators. Expressions may take up an entire line (72 characters) and cannot end with an operator.

All expressions are evaluated left-to-right. This means that an expression like,

```
A = 8 + 2 * 2 / 4 + 2
```

will not be evaluation like this:

```
A = 8 + ((2 * 2) / 4) + 2 = 13
```

but rather, like this:

```
A = (((8 + 2) * 2) / 4) + 2 = 7
```

When writing an expression, you should plan accordingly.

Logical expressions follow the form:

<variable> [<operator> <value>...]

where,

<operator> can be one of the relational operators (<, >, =<, =>, !=, ==) or one of two logical operators (&&, ||) defined above.

<value> can be either a variable or a constant.

A typical use of a logical expression is within a variable statement such as,

```
if .tcb_status == 02
```

where if a value of TCB_STATUS is equal to 2, the variable will be logically true and the statements following the IF statement will be executed.

7.3.5 Statements

Program statements can be divided logically into four groups:

- o DDL statements
- o System variables
- o System commands
- o Flow control commands

Each type is discussed in this section.

DDL Statements

DDL statements are of two types:

- o CDB Commands
- o DDL DATA Variables

You must define these commands and variables in advance using the Emulation Editor. This procedure is discussed in Section 4.

CDB Commands

Invoke any CDB names within the DDL currently defined as the target and they will be executed as commands.

NOTE

> Assign a DDL to the target address by using the Configure mode (see Section 5) or the .TARGET variable. Use .TARGET to redefine the CDB and DATA assignments (see Section 7.3.5).

Follow the CDB name by any number of expressions assigning various DDL TYPE field values. Each expression must be separated by a space.

The general form of the expression must conform to this convention:

[CDB name] [TYPE field] <assignment op> <value> [<op> <value>...]

where,

```
        <value> = constant or variable.
<assignment op> = any assignment operator.
           <op> = any operator excluding any relational operator
                  or the logical operators && and ||. (The
                  preferred value, however, is =.
```

For example, if the CCS DDL were active, the command line

```
READ ADDRESS = 0200
```

would set the ADDRESS field of the READ CDB to 200H and execute the command using that CDB. Results of the emulation would then be saved in the previously-assigned data buffer (SMALL is default). A default data length of 1 block is used unless otherwise stipulated.

You can assign as many variables per CDB name as you can fit on a line up to a maximum of 70 characters per line. For instance, if you had defined a WRITE CDB, you could invoke a command line looking like this:

```
WRITE ADDRESS = 0400  LENGTH = 0020
```

Where the address field is set to 400H and the transfer length is set to 20H. However, if you were to invoke this command,

```
WRITE LENGTH = 0020  LENGTH = 003F
```

only the last assignment of the transfer length value would be stored in the buffer, since the emulator would overwrite the first value.

To assign values to an enumerated field, invoke the enumerated field name as defined by the Command TYPE. If you defined a two-bit field within the CDB WRITE_BUFFER as 00 = HEAD_AND_DATA and 02 = DATA, then you could designate DATA by using this argument:

```
WRITE_BUFFER  DATA
```

This would set the data buffer in this enumerated field to 2. To set the other enumeration of this field, use this argument:

```
WRITE_BUFFER  HEADER_AND_DATA.
```

You can use both enumerations of the field in the same argument, but only the last enumerated value would actually be available in the buffer. The other value(s) would be overwritten.

If you have enumerated fewer than the allowable values per field in your Command TYPE definition, you can use the command line here to set additional values. For instance, in the case of the two-bit field within WRITE_BUFFER, there are only two values enumerated for four possible value entries: 00 = HEADER_AND_DATA and 02 = DATA. This means you can enumerate the 01 and 03 value of this field. To do this, override an existing enumerated value and set to another field value, as in this example:

```
WRITE_BUFFER  DATA = 03
```

In effect, this creates an enumerated value using the existing enumerated label of DATA. This does not effect the default value of DATA, merely creates an enumerated value for this one command only.

DDL DATA Variables

While CDB commands are determined by CDB TYPEs and COMMANDs you have previously defined, data commands are determined by DATA FORMAT and DATA definitions.

A data command is accessed from the currently active DDL, determined by the current target assignment and configuration table.

You can follow the data name by any number of expressions assigning various DDL FORM field values. The data name is separated from the data field and constants by a period (.).

The general form of the expression must conform to this convention:

Data_Name.[Data_Field_Name] <assignment op> <value> [<op> <value>..

where,

```
        <value> = constant or variable.
<assignment op> = any assignment operator.
           <op> = any operator excluding any relational operator
                  or the logical operators && and ||. (The
                  preferred value, however, is =.
```

For example, the command line

```
SENSED.SENSE_CODE = 04
```

would set the SENSE_CODE field of the SENSED buffer to 04H.

You can also assign a data field a constant value. For instance,

```
SENSED.LENGTH = 010
```

sets the LENGTH field in SENSED to 010H.

Constants can consist of decimals from 0 through 9, hexadecimals 0 through F, or strings enclosed in quotes ("string"). Maximum length of a string is 32 characters.

Assign values to as many variables as you require, but remember that data assignments follow the same rule as CDB assignments: only one statement is allowed per line. For instance, you might assign values to three fields within a SENSE buffer in this way:

```
SENSED.ERROR_CODE = 0F
SENSED.SENSE_CODE = 04
SENSED.FPV = 3
```

Here, error code is set to 0FH, the sense code is set to 04H, and FPV is set to 3.

However, if you were to define the same field twice, like this:

```
SENSED.ERROR_CODE = 0F
SENSED.ERROR_CODE = 02
```

only the last assignment of the error code value would be stored in the buffer, since the emulator would overwrite the first value.

To assign values to an enumerated field, simply invoke the enumerated field name as defined by the DATA TYPE. If you defined a four-bit field within the data buffer SENSED as 02 = NOT_READY and 03 = MEDIUM_ERROR, then you could designate NOT_READY by using this argument:

```
SENSED.NOT_READY = 004F
```

This would set the data buffer in this enumerated field to 0F. To test the other enumeration of this field, invoke MEDIUM_ERROR. You can use both enumerations of the field in the same argument, but only the last enumerated value would actually be available in the buffer. The other value(s) would be overwritten.

If you have enumerated fewer than the allowable values per field in your DATA TYPE definition, use the command line to set others. For instance, in the case of the four-bit field within SENSED, there are fourteen values enumerated for sixteen possible value entries -- 09 and 0F are still empty. To enumerate these entries, reassign an existing enumerated value to the empty field value, as in this example:

```
SENSED.NOT_READY = 09
```

In effect this creates a new enumerated value using the existing enumerated label of NOT_READY. This does not effect the default value of NOT_READY, merely creates a new enumerated value for this one command only.

System Variables

System variables control some aspect of the emulator set-up. Using these variables, you can reassign values to various parameters, such as target or LUN addresses, TCB variables, or block address numbers.

As with any variable, system variables can be set to a value or expression. They can either start a line, as in this example,

```
.CURSOR = CURSOR
```

or serve as a component within an argument on the line, as in this example:

```
If .TCB_STATUS != 0
```

Here is a list of the system variables:

.BLOCK	.LINK
.CURSOR	.LUN
.DISCONNECT	.OFFSET
.FILE	.PARITY
.FLAG	.TARGET
.INITIATOR	.TCB_n

To learn how these variables work, refer to the following pages.

NOTE

All system variables are preceded by a period (.), as in .FLAG or .OFFSET.

.BLOCK

Type:

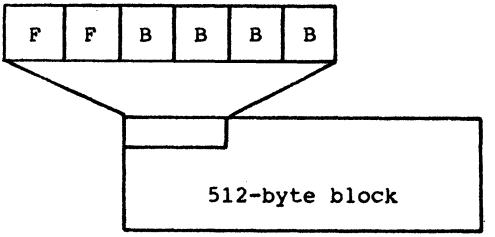
long

Function:

Use this variable to uniquely identify data "blocks" written from the 64K .LARGE buffer with a block number.

Comments:

This variable does housekeeping for storage of data in the LARGE buffer. If MARK is enabled, .FILE and .BLOCK bytes are inserted as the first 6 bytes of every 512-byte boundary written from the 64K .LARGE buffer. The value in .BLOCK will be incremented for each 512 byte block so that "unique" values can be placed in every 512-byte boundary written.



The first two bytes of the header are reserved for the file number. The next four bytes are reserved for the block number. For information on identifying the file number, see .FILE.

.CURSOR

Type:

int

Function:

Use this variable to set the cursor on the screen.

Comment:

Reload the cursor location on the screen using this command. Default location is upper lefthand corner (column 1, line 1). However, you can change this location by specifying the cursor location in several ways.

You can specify an relative position using the @ ("at") command, as in

```
@ 5,10
```

which tells the program to move the cursor position to line 5 column 10. See Program Commands in this section for more details.

You can also specify an absolute position by assigning a variable or expression, as in

```
.CURSOR = CURSOR
```

then attach values to this variable.

Be aware of the window you're working in. The cursor will move only in relation to a previous position and its present window.

.DISCONNECT

Type:

char

Function:

Use this variable to enable or disable the disconnect on the current target device. Default value is -1.

Comments:

Any non-zero value enables the disconnect.

[value] = 0 or non-0. If 0 then disconnect is off. If you specify a non-zero value (e.g. 1, 2, 3, and so on) the disconnect is enabled.

Invoke the disconnect to disable the message system. SCSI requires the message system be enabled (disconnect is disabled). SASI devices usually require that the message system is disabled (disconnect is enabled);

This function is also controlled by the CONFIGURE mode. See Section 5.

.FILE

Type:

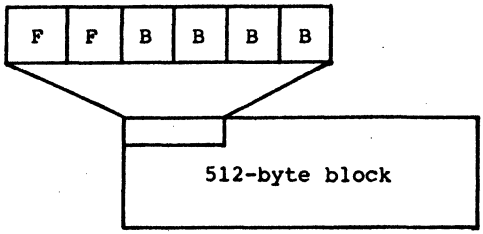
long

Function:

Use this variable to uniquely identify data "blocks" written from the 64K .LARGE buffer with a file number.

Comments:

This variable does housekeeping for storage of data in the LARGE buffer. If MARK is enabled, .FILE and .BLOCK bytes are inserted as the first 6 bytes of every 512-byte boundary written from the 64K .LARGE buffer. The value in .FILE will be incremented for each 512 byte block so that "unique" values can be placed in every 512-byte boundary written.



The first two bytes of the header are reserved for the file number. The next four bytes are reserved for the block number. For information on identifying the block number, see .BLOCK.

.FLAG

Type:

char

Function:

Use this variable to enable or disable the flag bit on the current target device. Default value is 0.

Comments:

Any non-zero value enables the flag bit.

[value] = 0 or non-0. If 0 then flag is off. If you specify a non-zero value (e.g. 1, 2, 3, and so on) the flag is enabled.

This function is also controlled by the CONFIGURE mode. See Section 5.

.INITIATOR

Type:

char

Function:

Use this variable to assign a SCSI device address to the current initiator. Default value is 7.

Comments:

Enter this command followed by an address number from 0 through 7.

This function is also controlled by the CONFIGURE mode. See Chapter 5.

.LINK

Type:

char

Function:

Use this variable to enable or disable the link bit for the current target device. Default value is 0.

Comments:

Any non-zero value enables the link bit.

[value] = 0 or non-0. If 0 then link is off. If you specify a non-zero value (e.g. 1, 2, 3, and so on) the link is enabled.

This function is also controlled by the CONFIGURE mode. See Section 5.

.LUN

Type:

char

Function:

Use this variable to assign the LUN field to the SCSI device address of the current target device driver. Default value is 0.

Comments:

Enter this command followed by an address number from 0 through 7. You can also control this function in the CONFIGURE mode. See Section 5.

.OFFSET

Type:

int

Function:

Use this variable to partition the .LARGE buffer. .OFFSET defines the starting address for the next data transfer from .LARGE.

Comments:

Use this variable if you are using the 64K .LARGE buffer. It automatically advances to the next 512 byte boundary after every .LARGE data transfer.

.PARITY

Type:

char

Function:

Use this variable to enable or disable the parity bit on the current target device. Default value is -1.

Comments:

Any non-zero value enables the parity function. Use [value] = 0 to disable parity.

This function is also controlled by the CONFIGURE mode. See Section 5.

.TARGET

Type:

char

Function:

Use this variable to assign a SCSI address to the target device. Default value is 0.

Comments:

Enter this variable followed by an address from 0 through 7.

This function is also controlled by the CONFIGURE mode. See Section 5.

.TCB_<parameter>

Function:

Use these variables to reference one of the task command block parameters.

Comments:

Reference a TCB_parameter by designating the parameter type, such as TCB_status or TCB_ID_MSG. This variable is used like any other variable except some cannot be written to.

Parameters:

For a complete list of TCB_parameters refer to Table A-4 of Appendix A.

Program Commands

Program commands control various aspects of the emulator and the program you are running on it.

These commands are listed below by their functional categories and discussed in more detail on the following pages.

Conditionals/Flow Control: commands that place conditions on the program or control program flow. (See Flow Control Commands section.)

BREAK	FOR... NEXT
CONTINUE	IF... ELSE... ENDIF
DO... UNTIL	NIF... ELSE... ENDIF
DO... WHILE	

I/O Control: commands that control input/output operations.

@	INPUT
@LPRINT	LPRINT
@PRINT	PRINT

State Analysis: commands that control the State Analyzer while in emulation.

ARM *
DISARM *
INDEX *

System: commands that control the emulator.

CLEAR *
HALT
RESET *
SUSPEND

Variable Declarations: commands that declare a variable type.

CHAR
INT
LONG
TEXT

* = used in both Interactive and Program mode.

ARM

Syntax:

ARM

Function:

Use this command to arm the Logic Analyzer. When invoked, the message ARMED is highlighted in the System Status window at the bottom of the screen.

Comments:

Use this function if you intend to capture data using the Logic Analyzer. To disarm the analyzer, use the DISARM command.

@

Syntax:

```
@ [<value> <value>]
```

Function:

Use this command to declare a position on the screen.

Comment:

When @ ("at") is placed alone on a line, this tells the emulator to clear the screen and place the cursor at the upper left-hand corner position (line 1, column 1), as in this example:

```
long block
int cursor
@
```

You can also specify an exact line and column location using this form:

```
@ line  column
```

as in this example:

```
long block
int cursor
@ 2 2
```

You can also specify the cursor position as a variable relative to the previous cursor position by using the .CURSOR command (see System Variables in this section).

CHAR

Syntax:

CHAR <variable name>

Function:

Use this command at the top of the program to declare variable(s) to follow as 8-bit data variables.

Comment:

Refer to Section 7.3.1 for a discussion of this command. Follow this command by one or more variable names. Each variable should be separated by a space or comma.

CLEAR

Syntax:

CLEAR [buffer] [value]

Function:

Use this command to clear data buffers.

Comments:

When you invoke this command, you clear all data from the specified buffer and prepare it for a new emulation. Default is the .LARGE buffer. Specify the .SMALL or DDL buffer.

You can also specify a pattern to the cleared buffer using an optional value. Default is cleared to zeros. However, by specifying a non-zero value, you create a pseudo-random sequence in the named buffer. (The pattern is repeated every 256 bytes.) For instance, a value of 2 gives an incremented sequence; 3 provides a decremented sequence. Any value greater than 7 adds the high 8 bits to the low 8 bits. (See the table below for details.)

CLEAR automatically clears .OFFSET and .FILE to zero.

Value	Byte Range
0	02201H
1	000FFH
2	01001H
3	0FFFFH
4	007FFH
5	089ABH
6	055AAH
7	055FFH

The low byte is placed into the specified buffer and the "high" byte is added to the "low byte"; the buffer address is then incremented.

DISARM

Syntax:

DISARM

Function:

Use this command to disarm the Logic Analyzer. When invoked, the message ARMED is not highlighted in the System Status window at the bottom of the screen.

Comments:

Use this function if you do not intend to capture data using the Logic Analyzer. To arm the analyzer, use the ARM command.

HALT

Syntax:

HALT

Function:

Use this command to halt the operation of the program.

Comments:

When invoked, this command stops the program.

INDEX

Syntax:

INDEX

Function:

Use this command to force the logic analyzer to save time and state data around some key index event.

Comments:

The command halts the capture operation after a specified number of writes to the acquisition memory, protecting data saved on either side of the INDEX command. When the analyzer is armed (see ARM and DISARM commands), time and state data is written in memory starting at address 000. At each write, the memory address register is incremented by 1. At a certain point the data will wrap back around on itself and begin to overwrite unless INDEX is invoked.

INDEX controls the capture process, loading the index count value, setting the counter to increment at every write, forcing a write to the memory when time and state data occurs, and halting capture when the index counter reaches its maximum count.

Once you have issued this command, the system status window highlights the word "INDEXED".

INPUT

Syntax:

INPUT ["prompt message string"] [variable]

Function:

Use this command to enter the value of a variable interactively while the program is running.

Comment:

When you invoke this command, the program stops and prompts you.

If there are no other arguments on the command line, the program stops and displays this message:

```
Press Any Key to Continue...
```

You can insert your own prompt by entering a text string after the command. (The text string must be enclosed in quotation marks, " ".) For instance,

```
INPUT "Text"
```

would cause the program to stop at a specific spot and display the prompt, "Text". You would then press any key to continue.

You can also insert a variable into the program. For instance,

```
INPUT variable_name
```

causes the program to stop and prompt you with this message:

```
?
```

Enter the variable name and press [Enter].

You can also combine a prompt string with a variable. For instance,

```
INPUT "Text" variable_name
```

causes the program to stop, display the prompt, and wait until you have entered a variable name and [Enter] before continuing.

INT

Syntax:

INT <variable name>

Function:

Use this command at the top of the program to declare the variable(s) to follow are 16-bit data variables.

Comment:

See 7.3.1 for details on using variable commands. Follow this command by one or more variable names. Each variable should be separated by a space or comma.

LONG

Syntax:

LONG <variable name>

Function:

Use this command to declare one or more 32-bit integers as variables.

Comments:

Follow this command with the variable names you wish to declare. Each variable should be separated by a space or comma.

LONG variables are more quickly processed by the emulator program since everything is converted to LONG before processing. However, LONG variables take up more space.

Refer to 7.3.1 for more details.

LPRINT

Syntax:

[@]LPRINT ["text string"] [[#]variable]

Function:

Use this command to instruct the program to print out a text string to a designated line printer.

Comments:

When you invoke this command, the program looks for a text string or a variable specified to the right of the command and prints it out to an output printer.

Printing begins at the start of the next line unless the @ prefix is used. This causes the string to be printed at the current cursor position.

Enclose any text string to be printed with quotation marks (" ") as discussed in Section 7.3.2.

Use the optional # sign to indicate you want the variable printed in hexadecimal. Default printing is decimal.

See PRINT also.

PRINT

Syntax:

```
[@]PRINT ["text string"] [[#]variable]
```

Function:

Use this command to instruct the program to print out a text string or variable to the screen.

Comments:

When you invoke this command, the program looks for a text string or variable to the right of the command and prints it to the screen as part of the program results.

Printing begins at the start of the next line unless the @ prefix is used. This causes the string to be printed at the current cursor position.

Enclose the text string to be printed with quotation marks (" ").

Use the optional # sign to indicate you want the variable printed in hexadecimal. Default printing is in decimal.

To print hardcopy, use the LPRINT command.

SUSPEND

Syntax:

SUSPEND

Function:

Use this command to suspend program operation.

Comment:

When you insert this command in a program it instructs the emulator to stop the program temporarily and return to the user.

If you are in Interactive mode, return to the suspended program by typing,

RUN [Enter]

If you are in the State Analysis program, return to the suspended program through menu selections by following these instructions:

1. Select the RUN option.

2a. Select the PROGRAM mode or,

2b. Select the INTERACTIVE mode and type

RUN [Enter]

TEXT

Syntax:

TEXT <variable name>

Function:

Use this command at the top of the program to declare the variable(s) to follow are 16-byte ASCII variables.

Comment:

See 7.3.1 for details on using variable commands. Follow this command by one or more variable names. Each variable should be separated by a space or comma.

Flow Control Commands

There are several commands used in the Program Mode enabling you to set conditions on the movement of the emulator through the program. Several of these commands enclose statements or series of statements. For the most part, they follow syntactical and grammatical conventions of BASIC.

The commands are discussed below.

BREAK

Syntax:

BREAK

Function:

Use this command to exit a program loop.

Comments:

Most program loops exit after a prescribed number of operations or tries. However, if certain conditions occur, it is often desirable to leave the loop prematurely. BREAK facilitates this by causing the program to immediately exit from the DO or FOR loop in which the command occurs. For example,

```
BLOCK = 0
DO
  READ ADDRESS = BLOCK
  IF .TCB_STATUS == 02
     BREAK
  ENDIF
  BLOCK += 100
WHILE BLOCK < 1000
```

This sample program would normally execute ten READ commands before satisfying the DO... WHILE loop. However, if .TCB_STATUS were 02 after any READ, the program immediately exits the DO... WHILE loop.

CONTINUE

Syntax:

CONTINUE

Function:

Use this command to advance a program loop.

Comments:

If you have specified a loop or routine within a program, then you can advance within this loop by using this command. For instance, in this case,

```
FOR A = 1 TO 5
  IF .TCB_STATUS != A
    CONTINUE
  ENDIF
  PRINT #A
NEXT
```

CONTINUE causes the program to go to the next value of A without printing its value as long as .TCB_STATUS is not equal to A.

DO... UNTIL

Syntax:

```
DO
.
.
.
UNTIL [logical expression]
```

Function:

Use this command to instruct the program to perform the operations specified in the body of the loop until a specific condition occurs.

Comment:

This command instructs the emulator to run one or more operations as set forth in the statements until the logical (non-zero) expression is met. For example,

```
t = 0
DO
  sense
  t += 1
UNTIL t == 100
```

This argument instructs the emulator to run the SENSE command until t reaches 100.

Nested Loops

DO... UNTIL loops may be nested; that is, a DO... UNTIL loop may be placed within another loop. When loops are nested, each loop should have a unique variable name in the logical expression. Also, the DO statement of the inside loop must appear before the UNTIL or end statement for the outside loop.

DO... WHILE

Syntax:

```
DO
.
.
.
WHILE [logical expression]
```

Function:

Use this command to instruct the program to perform the operations specified in the body of the loop while a specified condition exists.

Comment:

This command instructs the emulator to run one or several operations as set forth in the argument statements if or while logical expression is being met. For example,

```
t = 0
DO
 sense
 t += 1
WHILE t != 100
```

This argument instructs the emulator to run the SENSE command while t is not equal to 100.

Nested Loops

DO... WHILE loops may be nested; that is, a DO... WHILE loop may be placed within another loop. When loops are nested, each loop should have a unique variable name in the logical expression. Also, the DO statement of the inside loop must appear before the WHILE or end statement for the outside loop.

FOR... NEXT

Syntax:

```
FOR <variable> = x TO y [STEP z]
.
.
.
NEXT
```

Function:

Use this command statement to enable a series of instructions to be performed in a loop a given number of times.

Comment:

<Variable> is used as a counter. x is the initial value of the counter. y is the final value of the counter.

Lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP and checked to see if its value is now greater than the final value, y. If not, the program returns the statement after the FOR statement and the process is repeated. If the value is greater, the program execution continues with the statement following the NEXT statement. For example,

```
FOR block = 1 to 10 step 1
```

This indicates that you want the emulator to proceed through all possible conditions for the block, starting from 1 and proceeding to 2, 3, 4 and so forth until 10 is reached. The "to" value is included in the FOR... NEXT execution.

If STEP is not specified, the increment is assumed to be 1. If STEP is negative, the final value of the counter must be less than the initial value. The value of the counter decreases by increments each time through the loop until the counter value is less than the final value.

In specifying the step progress, you can stipulate steps like this:

step n	Increment upward through the range by n amount.
step -n	Progress backward through the range.
step var	Use the value of the variable as the STEP amount.

Nested Loops

FOR... NEXT loops may be nested; that is, a FOR... NEXT loop may be placed within another loop. When loops are nested, each loop should have a unique variable name. Also, the NEXT statement of the inside loop must appear before the NEXT or end statement for the outside loop.

IF...ELSE... ENDIF

Syntax:

```
IF <logical expression>
.
.
[ELSE
.
.     ]
ENDIF
```

Function:

Use this command to set conditions for the running of a particular operation based on the value of specified variables.

Comments:

Stipulate conditions after the IF command. The condition is written in the form of a logical expression. Follow the condition by any statements you want the emulator to run, if the logical expression is non-zero.

Use the optional ELSE command to specify any command or procedure you might want the emulator to run, given the logical expression in the IF statement evaluates to zero.

End the IF... ELSE argument with ENDIF. ENDIF occupies its own line.

For example,

```
IF .TCB_STATUS == 0
  READ_USAGE_COUNTERS
  PRINT "BLOCKS READ=" COUNTERS.BLOCK_READ
  PRINT "RECOVERABLE ERRORS=" COUNTERS.RECOVERED_ERRORS
ELSE
  PRINT "CHECK CONDITION"
ENDIF
```

This statement instructs the emulator that if TCB_STATUS is 0, the DDL command READ_USAGE_COUNTERS is sent out to the target and two buffer fields, COUNTERS.READ_ERRORS and COUNTERS.RECOVERED_ERRORS, are displayed with appropriate labels.

If the TCB_STATUS block is non-zero, then the emulator is instructed to display the message, "Check Condition."

Nested IF statements

IF... [ELSE...] ENDIF constructs may be nested. That is, an IF... ENDIF construct can be placed within loop construct. However, the ENDIF statement of the inside loop must appear before the ENDIF, UNTIL, WHILE, or NEXT statement of the outside loop. They may not overlap.

NIF...ELSE... ENDIF

Syntax:

```
NIF <logical expression>
.
.
[ELSE
.
.     ]
ENDIF
```

Function:

Use this command statement to set conditions for the running of a particular operation based on the value of specified variables. This statement is the explicit opposite of the IF... ELSE statement.

Comments:

NIF literally means "not if" and can be stated using the IF command. For example,

```
NIF a == 1
```

is equivalent to,

```
IF  a != 1
```

NOTE

When testing a variable for equality to a constant, the constant must appear on the right side of the argument. Therefore, 1 == A is wrong; A == 1 is the correct form.

Stipulate conditions after the NIF command. The condition is written in the form of a logical expression. Follow the condition by the procedure(s) or command(s) you want the emulator to run if the logical expression is zero.

End the NIF... ELSE argument with ENDIF. ENDIF occupies its own line.

For example,

```
NIF .TCB_STATUS == 2
  READ_USAGE_COUNTERS
  PRINT "BLOCKS READ=" COUNTERS.BLOCK_READ
  PRINT "RECOVERABLE ERRORS=" COUNTERS.RECOVERED_ERRORS
ELSE
  PRINT "CHECK CONDITION"
ENDIF
```

This statement instructs the emulator that in the event that .TCB_STATUS is not equal to 2, the DDL command READ_USAGE_COUNTERS is sent out to the target and two buffer fields, COUNTERS.READ_ERRORS and COUNTERS.RECOVERED_ERRORS, are displayed with appropriate labels.

If the TCB_STATUS block is equal to 2, then the emulator is instructed to display the message, "Check Condition."

Nested NIF statements

NIF... [ELSE...] ENDIF constructs may be nested. That is, a NIF... ENDIF construct can be placed within loop construct. However, the ENDIF statement of the inside loop must appear before the ENDIF, UNTIL, WHILE, or NEXT statement of the outside loop. They may not overlap.

Table 7-1. Program Command Summary

System Variables

Variables	Type	Function
.BLOCK	Long	Identifies 512-byte data "blocks" written from the 64K .LARGE buffer with a block number.
.CURSOR	Int	Sets the cursor on the screen.
.DISCONNECT	Char	Enables or disables the disconnect on the current target device.
.FILE	Int	Identifies data "blocks" written from the 64K .LARGE buffer with a file number.
.FLAG	Char	Enables or disables the flag bit on the current target device.
.INITIATOR	Char	Assigns a SCSI address to the current initiator.
.LINK	Char	Enables or disables the link bit on the current target device.
.LUN	Char	Assigns the LUN field to the SCSI device address of the current target device driver.
.OFFSET	Int	Partitions the .LARGE buffer into address blocks.
.PARITY	Char	Enables or disables the parity bit on the current target device.
.TARGET	Char	Assigns a SCSI address to the current target device.
.TCB_x	Int	Sets a designated task command block parameter and changes the TCB image of the current target device. For a complete list of .TCB parameters, see Table A-4 of Appendix A.

Table 7-1. Program Command Summary (continued).

Program Commands

Command	Function
ARM	Arms the Logic Analyzer.
@	Declares a position on the screen or prefixes PRINT or LPRINT.
CHAR	Declares variable(s) as 8-bit signed variables.
CLEAR	Clears designated data buffers.
DISARM	Disarms the Logic Analyzer.
HALT	Halts the operation of the program.
INDEX	Forces the Logic Analyzer to save time and state data around some key index event.
INT	Declares variable(s) as 16-bit signed variables.
INPUT	Allows entry of a variable value interactively while the program is running.
LPRINT	Instructs the program to print out a text string to a designated line printer at start of next line. @LPRINT causes string to be printed at current cursor position.
LONG	Declares variable(s) as 32-bit signed variables.
PRINT	Instructs the program to print out a text string or variable to the screen. @PRINT causes string to be printed at current cursor position.
SUSPEND	Suspends program operation.
TEXT	Declares variable(s) as 16-byte ASCII variables.

Table 7-1. Program Command Summary (continued).

Flow Control Commands

Command	Function
BREAK	Exits a program loop.
CONTINUE	Advances a program loop.
DO.. UNTIL	Instructs the program to perform specified operations in the body of the loop until a specific condition occurs.
DO.. WHILE	Instructs the program to perform specified operations in the body of the loop while a specified condition exists.
FOR.. NEXT	Causes a series of instructions to be performed in a loop a given number of times.
IF.. ELSE.. ENDIF	Sets conditions for the running of a particular operation based on the value of specified variables.
NIF..ELSE.. ENDIF	The explicit opposite of the IF.. ELSE construct.

7.4 Initiating the Program

To run a program, follow this procedure:

STEP 1
Return to the main Initiator Emulation menu.

(If you are still in the program, press [Esc] several times until you see this menu.)

STEP 2
Select RUN from the options. Press [Enter].

The following prompt appears below the SCSI Bus Status window:

```
Select RUN mode [INTERACTIVE  PROGRAM]: _
```

STEP 3
Select PROGRAM. This prompt appears:

```
Location of RUN program [MEMORY DISK]: _
```

STEP 4
Select either memory or disk depending on whether the program you want resides in the computer memory or on diskette. If you select memory, follow step 5a; if you select diskette, follow step 5b.

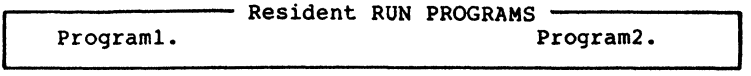
STEP 5a
If you select memory, the following window and prompt appears:

```
┌──────── Resident Device Description Library(s) ────────┐
 Library1                              Library2
└────────────────────────────────────────────────────────┘

Enter name of desired Library:
```

Press [Down] to select the DDL that contains the program you want to run, or enter a DDL library name at the prompt. Press [Enter].

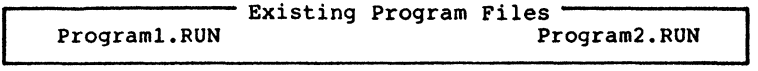
The following window and prompt appears:



Enter name of desired RUN file:

Press [Down] to select the program you want to run, or enter a new program name at the prompt. Press [Enter].

STEP 5b
If you select diskette, the following window and prompt appears:



Enter name of desired RUN file:

Press [Down] to select the program you want to run, or enter a new program name at the prompt. Press [Enter].

If no program exists on the disk, then the program returns to the initial RUN menu:

Press [Down] to select the resident DDL file you want to run, or enter a new library name at the prompt. Press [Enter]. If the DDL library you entered is valid, you are returned to the previous window and prompted for a program. Select one of the resident run programs.

STEP 6
Once you have selected a run program, the program is initiated. When the emulator is finished running the program, it displays this message:

Program execution complete, hit any key to resume

STEP 7
Press any key and you are returned to the main Initiator Emulation menu.

7.5 Sample Programs

To help you in learning the syntax of this program, several sample programs are presented for your review.

NOTE

To assist in describing each program line, line numbers are pictured in these examples. Line numbers are not required for the actual programs.

Sample 1

Below is a program designed to test cursor position and print entry.

Program:

```
1    char repeat
2    int spot

3    @ ;clear screen
4    @print "Start cursor test"
5    spot = .cursor ;save cursor location
6    @print #.cursor
7    print
8    input "Start spot test" repeat
9    .cursor = spot ;restore cursor
10   @print repeat "that's all folks"
```

Explanation:

1-2 On the first two lines the variables are declared. Repeat is defined as an 8-bit variable type while spot is identified as a 16-bit integer. Note that the system variable .CURSOR is an integer type.

3 On the third line the @ command clears the screen. (The comment line following this command describes the action.)

4 The next line invokes the @print command which tells the emulator to print the string "Start cursor test" to the screen, starting at the current cursor position.

5 The variable 'spot' is then assigned the value of the system variable .CURSOR. This causes the variable to "save" the current cursor location. (The comment line once again underlines this.)

6 The current cursor position is then displayed on the screen in hexadecimal form (remember: # instructs the emulator to output in hex).

7 'Print' instructs the emulator to do a carriage return and line feed, moving the cursor to the first position on the next line.

8 'Input' instructs the emulator to place a prompt "Start spot test" and this is associated with the character variable 'repeat'.

9 The cursor is then restored by assigning the system variable .CURSOR the value of CURSOR.

10 The value of the variable 'repeat' is printed on the screen followed by the words "That's all folks" starting at the present cursor position.

Sample 2

This is a program designed to read 0780H blocks of data and display the current and last address of the address field in the DDL command READ.

Program:

```
1    int a
2    @
3    for a = 0 to 0780 step 080
4      read address = a  length = 080
5      if .TCB_GATE
6       break
7         else
8         print "Current address =" #a
9         endif
10   next
11   print "Last address =" #a
```

Explanation:

1 This declares "a" as a 16-bit integer variable.

2 The screen is cleared.

3-10 FOR.. NEXT loop wherein conditions of the emulation are set out as explained below.

3 This instructs an iteration of the loop for variable 'a' from value 0 to 0780. The emulator is alerted to hex ranges by a prefix 0. The command word 'step' instructs the emulator to proceed through the loop in 080 increments.

4 Given this range and interval, 'a' is now assigned to the address field in the DDL command READ. At the same time, the length field is defined as 080.

5-6 The condition is placed on this simulation that if the parameter .TCB_GATE is non-zero, the program is to break to line 11 immediately.

7-8 Otherwise, indicated by 'else', if .TCB_GATE is zero, the emulator is to display the string "Current address =" on the screen at the default position followed by the value of 'a' at that moment, outputted in hex form.

9-10 This ends the FOR.. NEXT loop and instructs the emulator to keep doing this loop ('next') in 080 increments until the upper limit, 0780, is exceeded.

Sample 3

This sample causes the target to display information about its name, serial number, and the capacity of the device.

Program:

```
1    long size
2    int cursor

3    @ 6 2

4    arm    ;arm analyzer
5    test_unit_ready    ;wake up target

6    if .tcb_status == 02    ; check condition
7        request_sense
8        print " Sense Requested"
9    endif
10   clear inquired
11   clear capacity
12   read_capacity  ;get target info
13   inquiry
14   cursor = .cursor
15   @print inquired.model
16   .cursor = cursor
17   @print " model:"
18   print " Serial Number:" inquired.serial_number
19   size = capacity.last_block
20   size += 1
21   size *= capacity.block_size
22   print  " Capacity:" size
23   print  " bytes with" capacity.block_size
24   @print " bytes/block"
```

Explanation:

1-2 The variables size and cursor are declared -- one as long, the other as integer.

3 The screen is cleared and the cursor is positioned at line 6 column 2.

4 The analyzer is armed.

5 The target is awakened with a DDL command.

6-9 If a "check condition" status is returned by the target, the REQUEST_SENSE command is sent to the target and the message "Sense Requested" is sent to the screen.

10 The buffer INQUIRED is cleared.

11 The buffer CAPACITY is cleared.

12 Once these buffers are cleared, the command READ_CAPACITY is invoked followed by INQUIRY. Data from these commands is deposited in the cleared buffers.

13 The string "Manufacture:" is followed by the value of the INQUIRED.MANUFACTURE buffer field from the INQUIRY command.

14 The screen position is saved.

15 The buffer field INQUIRED.MODEL is printed at the former cursor point.

16 The cursor is restored to the screen at the point just previous to the last buffer field.

17 The string " Model:" is displayed on the screen at the new cursor point.

18 The string "Serial Number:" is displayed on the screen followed by the value from the buffer INQUIRED.SERIAL_NUMBER.

19 The size variable is equated to the data buffer field CAPACITY.LAST_BLOCK.

20 The size variable is incremented by 1.

21 Size is multiplied by the CAPACITY.BLOCK_SIZE data field.

22 The resulting value of size is displayed after the string "Capacity =".

23 The "Capacity =" string is followed by the another string, "bytes with", followed by the value of the CAPACITY.BLOCK_SIZE data field.

24 At the last cursor position, the string "bytes/block" is displayed.

The screen produced by this program should look something like this:

```
Manufacture: xxxxxx

Model: yyyyyyy

Serial Number: zzzzzz

Capacity: nnnnn bytes with aaaaa bytes/block
```

1. Task Control Primitives

Table A-1 contains a list of all Task Control primitives currently defined for the Initiator Emulation. Hex values associated with each primitive appear after the primitive notation, i.e. $00H. Access these values using either TCB_PRIMITIVE or VIEW in Interactive mode.

Table A-1. Task Control Primitives.

Primitives	Value	Description
T_OPEN	01	Opens a task, using the current INITIATOR, TARGET, and LUN IDs.
T_CLOSE	05	Closes the task.
T_SET_UP	09	This converts the TCB_DATA_OFFSET and TCB_DATA_SEGMENT into TCB_DMA pointer format.
T_SAVE	0B	This transfers the TCB_DMA pointers into the TCB_SAVE pointers.
T_RESTORE	0D	This transfers the TCB_SAVE pointers into the current TCB_DMA pointers.
T_STATUS	0F	This places an image of the SCSI data bus into TCB_MESSAGE[0], while the TCB_MESSAGE[1] contains the BUS_STATUS_REG and TCB_MESSAGE[2] contains the BUS_STATUS image. BUS_STATUS_REG and BUS_STATUS are registers within the SCSI hardware interface.

2. Extreme Primitives

Use the extreme drive primitives listed in Table A-2 to assert or deassert a single SCSI bus control signal or transfer a single byte across the bus. Although the following commands may be used to implement normal SCSI transactions, they are intended for negative testing and don't check for the validity of the desired operation.

Table A-2. Extreme Primitives.

Primitive	Value	Description
P_SET_ACK	13*	This asserts ACK on the SCSI bus.
P_CLR_ACK	15*	This deasserts ACK on the SCSI bus.
P_SET_ADR	17	This sets the SELECTION address register in the SCSI controller to the value in TCB_MESSAGE[0]. The bit position of any address the chip should respond to should be set to one.
P_SET_ARB	19	This asserts ARB on the SCSI bus.
P_CLR_ARB	1B	This deasserts ARB on the SCSI bus.
P_SET_ATN	1D*	This asserts ATN on the SCSI bus.
P_CLR_ATN	1F*	This deasserts ATN on the SCSI bus.
P_SET_BSY	21	This asserts BSY on the SCSI bus.
P_CLR_BSY	23	This deasserts BSY on the SCSI bus.
P_SET_BUS	25	This asserts the DATA on the SCSI bus.
P_CLR_BUS	27	This deasserts the DATA on the SCSI bus.
P_SET_DATA	2D	This sets the DATA register in the SCSI controller with the value in TCB_MESSAGE[0]. If the BUS is enabled, the data will appear on the SCSI bus.

Table A-2. Extreme Primitives (continued).

Primitive	Value	Description
P_SET_RST	3B*	This asserts RST on the SCSI bus.
P_CLR_RST	3D*	This deasserts RST on the SCSI bus.

* = Initiator-specific primitive. All others can be used by both Initiator and Target emulators.

3. Conventional Primitives

This is a list of all conventional primitives currently defined for the Initiator Emulation. Values associated with each primitive appear after primitive notation.

Table A-3. Conventional Primitives.

Primitive	Value	Description
P_MSG_R	00	This sets ATN and provides TCB_ACT_COUNT with the number of message bytes received.
P_MSG_W	02	This sets ATN and waits for the target to go into a message out phase. Use the primitive with a message in the form: P_MSG_W [message] where [message] is one of the SCSI messages discussed in part 7. If no message is appended to the primitive, TCB_MESSAGE[0] is sent. (Default only supplied in Program mode; you must supply your own message name in Interactive.)
P_STAT_R	04	Returns with TCB_STAT byte if successful. Or TCB_GATE will have exception code.
P_DATA_R	08	Receives TCB_DATA_OFFSET/_SEGMENT, TCB_COUNT_SPC data if successful. Or TCB_GATE will have exception code. TCB_COUNT_ACT contains the number of bytes transferred.
P_DATA_W	0A	Sends TCB_DATA_OFFSET/_SEGMENT, TCB_COUNT_SPC data if successful. Or TCB_GATE will have exception code. TCB_COUNT_ACT contains the number of bytes transferred.

Table A-3. Conventional Primitives (continued).

Primitive	Value	Description
P_CMD_W	0E	Sends TCB_CDB if successful. Or TCB_GATE will have exception code. This primitive requires a CMD name to execute. Field names are optional. Thus: P_CMD_W [CMD_name] [field_nm=val..] Otherwise, if no CMD name is provided, the system will use whatever values are already in the TCB_CDB. (Default only supplied in Program mode; you must supply your own field names in Interactive.)
P_DISC	10	Sets driver internally tc disconnect.
P_SELT	12	The desired target will be selected.
P_RSEL	14	Enables a target to reselect the initiator.
P_RESET	16	This issues a reset message to the target if possible. Optionally the initiator can reselect the target and LUN to force a reset.
D_SELT	22	Like P_SELT only sends IDENTIFY_ MESSAGE as well.
D_RSEL	24	Like P_RSEL only accepts IDENTIFY_ MESSAGE as well.
D_DATA_R	26	This functions as P_DATA_R with the additional capability to deal with disconnect reselect automatically.
D_DATA_W	28	This functions as P_DATA_W with the additional capability to deal with disconnect reselect automatically.

Table A-3. Conventional Primitives (continued).

Primitive	Value	Description
D_CMD_W	32	Functionally, this is a composite of P_SELT and P_CMD_W. This primitive requires a CMD name to execute. Field names are optional. Thus: D_CMD_W [<CMD_nm> [Field_nm=val..]] If no field names are appended, the system will use whatever values are currently available. (Default only supplied in Program mode; you must supply your own field names in Interactive.)
D_TERM	34	The TCB_STAT receives the status byte and TCB_MESSAGE the completion byte.
D_READ	3A	Functional composite of D_CMD_W and D_DATA_R. This primitive requires a CDB name to execute. Field values are optional. Thus: D_READ [<CDB_nm> [field_nm=val..] [data_buffer]] If no field names are appended, the system will use whatever values are currently available. (Default only supplied in Program mode; you must supply your own field names in Interactive.) If CDB name is provided, all parameters are set automatically. If not, in addition to setting TCB_CDB, this command requires you to set TCB_COUNT_SPCD, TCB_DATA_OFFSET, and TCB_DATA_SEG.

Table A-3. Conventional Primitives (continued).

Primitive	Value	Description
D_WRITE	3C	Functional composite of D_CMD_W and D_DATA_W. This primitive requires a CDB name to execute. Field values are optional. Thus: D_WRITE [<CDB_nm> [field_nm=val..] [buffer]] If no field names are appended, the system will use whatever values are currently available. (Default only supplied in Program mode; you must supply your own field names in Interactive.) If CDB name is provided, all parameters are set automatically. If not, in addition to setting TCB_CDB, this command requires you to set TCB_COUNT_SPCD, TCB_DATA_OFFSET, and TCB_DATA_SEG.
D_INITIATOR	3E	Functional composite of D_CMD_W and D_DATA_R or D_DATA_W and D_TERM. This primitive requires a CDB name to execute. Field values are optional. Thus: D_INITIATOR [<CDB_nm> [field_nm=val..] [buffer]] If no field names are appended, the system will use whatever values are currently available. (Default only supplied in Program mode; you must supply your own field names in Interactive.) If CDB name is provided, all parameters are set automatically. If not, in addition to setting TCB_CDB, this command requires you to set TCB_COUNT_SPCD, TCB_DATA_OFFSET, and TCB_DATA_SEG.

4. Control Structures

Table A-4 lists the Task Control Block variables. The type of each variable is given and the variable itself is defined.

Table A-4. Control Structures.

Variable	Type	Definition
TCB_GATE*	char	Gate byte, set by driver.
TCB_PRIMITIVE	char	Primitive code such as P_MSG_R.
TCB_STATE	int	State of SCSI transaction.
TCB_ID_MSG	char	ID message sent or received.
TCB_CDB[12]	char array	SCSI CDB for task.
TCB_STATUS	char	Status byte sent or received.
TCB_COMPLETION	char	Completion message for task.
TCB_COUNT_SPCD	long	Item count requested, 24-bit capability.
TCB_COUNT_ACT	long	Item count actually transferred.
TCB_DATA_OFFSET	int	Buffer memory address in data segment.
TCB_DATA_SEGMENT	int	Data segment to use
Current data pointers used internally by driver for housekeeping:		
TCB_DMA_HIGH	char	Contents of DMA_HIGH.
TCB_DMA_MODE	char	Contents of DMA_MODE.
TCB_DMA_ADDRESS	int	Contents of DMA_ADR.
TCB_DMA_COUNT	int	Contents of DMA_CNT.
TCB_ACT_COUNT	int	Difference at EOP.

Table A-4. Control Structures (continued).

Variable	Type	Definition
Driver task_control context:		
TCB_INITIATOR_ID	char	SCSI initiator address.
TCB_TARGET_ID	char	SCSI target address.
TCB_LUN_ID	char	SCSI LUN address.
TCB_OPTION*	int	Working copy.
TCB_BLK_SZ	int	Block size unless = 0.
TCB_MESSAGE[8]	char array	Transient message space large enough for extended messages.
Current and saved data pointers, used by save and restore pointers:		
TCB_CURRENT_ADDR*	int	DMA image address.
TCB_CURRENT_HIGH*	char	DMA image high nibble.
TCB_CURRENT_COUNT*	int	DMA image count.
TCB_CURRENT_PHASE*	char	DMA TCB image.
TCB_CURRENT_MODE*	char	DMA MODE image.
TCB_SAVED_ADDR*	int	DMA image address.
TCB_SAVED_HIGH*	char	DMA image high nibble.
TCB_SAVED_COUNT*	int	DMA image count.
TCB_SAVED_PHASE*	char	DMA TCR image.
TCB_SAVED_MODE*	char	DMA MODE image.

Table A-4. Control Structures (continued).

Variable	Type	Definition
TCB_INTERRUPT_LOCATION*	int	Where to go on interrupt.
TCB_RETURN_LOCATION*	int	Where to go after interrupt.
TCB_LINK_OFFSET	int	Next TCB in linked command.
TCB_LINK_SEGMENT	int	Next TCB in linked segment.
TCB_ASSIGNMENT*	int	Driver fills this in.
TCB_GROUP*	int	Used by driver.

* = This command belongs to the driver. You can't write into it. For example, writing

```
.TCB_GROUP = 12
```

would produce an error message.

5. TCB Flags

Certain bits within the TCB structure provide you with valuable information about the function of the system. To read these bits, see Table A-5.

Table A-5. TCB Flags.

Flags	Definitions
TCB_GATE bits:	
GATE_OWNER_FLAG	bit 7 denotes ownership 1 = driver, 0 = application
TCB_OPTION bits:	
EXTRA_DATA_FLAG	bit 8 = throw away extra data 1 = valid, 0 = invalid
RD_FLAG	bit 7 + 6 = transfer mode
WR_FLAG	00 = any transfer 01 = read only 10 = write only 11 = no transfer
TIME_OUT_FLAG	bit 5 = timeout 1 = valid, 0 = invalid
MESSAGE_FLAG	bit 4 = use message system 1 = valid, 0 = invalid
DISCONNECT_FLAG	bit 3 = disconnect 1 = valid, 0 = invalid
PARITY_FLAG	bit 2 = parity 1 = valid, 0 = invalid
INITIATOR_FLAG	bit 1 = device mode 0 = target, 1 = initiator

6. TCB Error Codes

If, after a driver primitive is executed, TCB_GATE is not zero (COMMAND COMPLETE), it will contain one of these error codes:

Table A-6. Error Codes.

Code Name	Value	Meaning
SELECTION_ERROR	1	Selection failed status.
DISCONNECTED	2	Disconnected status.
PARITY_ERROR	3	Parity error status.
BUS_RESET	4	SCSI bus reset status.
CHIP_FAILURE	5	Chip failure status.
MESSAGE_BYTE	6	Message in byte being returned.
DIFFERENT_PHASE	7	Unexpected phase requested.
ILLEGAL_CMD	8	Conflicting or unknown requests.
OPERATOR_ABORT	9	You aborted this operation.
TASK_CONFLICT	12	Fields within T_OPEN DIS conflict with prior T_OPEN.
TASK_INACTIVE	13	Specified task is inactive.
NO_TASK_SPACE	14	No TCB's remaining.
BUS_NOT_FREE	15	SCSI bus appears to be hung.

7. SCSI Message System

Table A-7 lists SCSI messages together with their value and the signal direction (where IN indicates target was originator; OUT indicates initiator was originator).

Table A-7. SCSI Message System.

Message	Value	Direction	Meaning
COMMAND_COMPLETE	0	IN	
ABORT	6	OUT	Command aborted.
MESSAGE_REJECT	7	IN/OUT	Message was rejected.
NO_OPERATION	8	OUT	This is a dummy message.
BUS_DEVICE_RESET	0CH	OUT	Device on the bus was reset.
DISCONNECT_MESSAGE	4	IN	Message was disconnected.
IDENTIFY_MESSAGE	80H	IN/OUT	Identification of the TARGET/LUN.
SAVE_DATA_POINTERS	2	IN	Data pointers were saved.
RESTORE_POINTERS	3	IN	Data pointers were restored.
INIT_DETECT_ERROR	5	OUT	Errors were detected at initialization.
MSG_PARITY_ERR	9	OUT	A message parity error was detected.
LINKED_CMD_CMPL	0AH	IN	Linked COMMAND COMPLETED.
LNKD_CMD_CMPL_FLG	0BH	IN	Linked COMMAND COMPLETE flag.

APPENDIX B - SCSI-2 DIRECT ACCESS DDL

INTRODUCTION

This appendix is a catalog of the commands defined in the Device Description Library provided on the program disk as file SCSI2_DA.DDL. This DDL file is based on a preliminary draft of the proposed SCSI-2 standard dated October 31, 1986. This rendition of the SCSI-2 direct access DDL must be considered only a prototype which requires editing by you to match the specific implementation of your particular target. To do this you will need to refer to the SCSI manual provided by the target vendor. Presently the proposed standard is in considerable flux and is undergoing changes by, and at the sole discretion of, the ANSI X3T9.2 Task Group.

Table B-1 provides an alphabetical list of the commands in the catalog. Each catalog entry includes the template of the CDB, the command name, the referenced TYPE definition, the CDB data as a string of hex digits, the name of each defined field in the CDB and the DATA buffer attached to the command (refer to the example COPY command below).

Each bit of the CDB template is coded. "Operation code" identifies the first byte of each CDB. Its assigned value is to the right of the template. The target LUN address is inserted in the "Unit#" field when the command is issued to a target. A string of a lower case letter identifies a named field. The name assigned each field can be found to the right of the template. A string of an upper case letter identifies an enumerated field. The name of each enumeration is listed to the right of the letter in ascending order, separated by comas. Unnamed enumerations appear as blanks. For example, the enumerated value 1 of field A of the COPY CDB has been assigned the label PAD while the value 0 has been assigned no label. Bit positions not assigned to a field are identified in the template by 0.

If there is a DATA buffer assigned to a CDB other than the normal default buffer .SMALL, it is identified to the right of the template.

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: COPY
000         Operation  code    CDB type: TYPE_004
001         Unit# 0 0 0 0 A    CDB data: 180000001000
002         b b b b b b b b    Operation code = 18h
003         b b b b b b b b    A: , PAD
004         b b b b b b b b    bb ... bb: LENGTH
005         0 0 0 0 0 0 F+L    DATA buffer: COPY_4
```

As mentioned above, you will need to edit this DDL to match the implementation of your particular target. We suggest you save any changes under a file name different than SCSI2_DA. Below are some additional hints (cans and can'ts) for you to consider as you edit the DDL.

First, you CAN:

create a new FORM or TYPE field by assigning bits presently unassigned to the new field,

change the name of a TYPE or FORM field,

change the name of a CDB or DATA buffer,

change the DATA buffer attached to a CDB.

Second, you CAN, BUT not without impact elsewhere:

change the size or kind of a field. If a resized field encroachs into another defined field, that other field is deleted.

change the size of a TYPE template. If you do you must delete and recreate all CDB's referencing the changed template.

change the size of a FORM template. If you do you must delete and recreate all DATA buffers referencing the changed template.

rename or delete a TYPE or FORM template. If you do you must delete and recreate all CDB's or DATA buffers referencing the renamed or deleted template.

delete a TYPE or FORM field. If you do all subsequent fields in the TYPE or FORM will be deleted.

Lastly, you CAN'T:

change the TYPE of a CDB (You must delete the existing CDB and recreate it with the new TYPE assigned),

change the FORM of a DATA buffer (You must delete the existing DATA and recreate is with the new FORM assigned).

Long names for CDB's, DATA buffers and FORM and TYPE fields should be avoided. They do impact program performance,

increase DDL size and require more typing at the keyboard. For understandability, the names used in this DDL are particularly verbose. You may wish to edit these names to something equally meaningful to you while reducing their length.

Keep in mind that any edits you make on a TYPE or FORM template (other than field name changes) may adversely impact the CDB's and DATA buffers referencing those templates.

Table B-1 Commands for Direct Assess Devices

TEST UNIT READY

```
BYTE -- bit 7-6-5-4-3-2-1-0
000         Operation  code
001         Unit# 0 0 0 0 0
002         0 0 0 0 0 0 0 0
003         0 0 0 0 0 0 0 0
004         0 0 0 0 0 0 0 0
005         0 0 0 0 0 0 F+L
```

```
CDB name: TEST_UNIT_READY
CDB type: TYPE_001
CDB data: 000000000000
Operation code = 00h
```

REZERO UNIT

```
BYTE -- bit 7-6-5-4-3-2-1-0
000         Operation  code
001         Unit# 0 0 0 0 0
002         0 0 0 0 0 0 0 0
003         0 0 0 0 0 0 0 0
004         0 0 0 0 0 0 0 0
005         0 0 0 0 0 0 F+L
```

```
CDB name: REZERO_UNIT
CDB type: TYPE_001
CDB data: 010000000000
Operation code = 01h
```

REQUEST SENSE

```
BYTE -- bit 7-6-5-4-3-2-1-0
000         Operation  code
001         Unit# 0 0 0 0 0
002         0 0 0 0 0 0 0 0
003         0 0 0 0 0 0 0 0
004         a a a a a a a a
005         0 0 0 0 0 0 F+L
```

```
CDB name: REQUEST_SENSE
CDB type: TYPE_002
CDB data: 030000000000
Operation code = 03h
aaaaaaaa: LENGTH
DATA buffer: SENSED
```

FORMAT UNIT

```
BYTE -- bit 7-6-5-4-3-2-1-0
000         Operation  code
001         Unit# A B c c c
002         0 0 0 0 0 0 0 0
003         d d d d d d d d
004         d d d d d d d d
005         0 0 0 0 0 0 F+L
```

```
CDB name: FORMAT_UNIT
CDB type: TYPE_007
CDB data: 040000000000
Operation  code = 04h
A: , FMTDATA
B: , CMPLST
ccc: LIST_FORMAT
dd ... dd: INTERLEAVE
DATA buffer: DEFECTS
```

REASSIGN BLOCKS

```
BYTE -- bit 7-6-5-4-3-2-1-0
000         Operation  code
001         Unit# 0 0 0 0 0
002         0 0 0 0 0 0 0 0
003         0 0 0 0 0 0 0 0
004         0 0 0 0 0 0 0 0
```

```
CDB name: REASSIGN_BLOCKS
CDB type: TYPE_001
CDB data: 070000000000
Operation code = 07h
DATA buffer: DEFECT_LIST
```

```
 005          0 0 0 0 0 0 F+L
```

READ (6 byte CDB)

```
 BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: READ
 000         Operation  code   CDB type: TYPE_008
 001         Unit# a a a a a   CDB data: 080000000000
 002         a a a a a a a a   Operation code = 08h
 003         a a a a a a a a   aa ... aa: ADDRESS
 004         b b b b b b b b   bbbbbbbb: LENGTH
 005         0 0 0 0 0 0 F+L
```

WRITE (6 byte CDB)

```
 BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: WRITE
 000         Operation  code   CDB type: TYPE_008
 001         Unit# a a a a a   CDB data: 0A0000000000
 002         a a a a a a a a   Operation code = 0Ah
 003         a a a a a a a a   aa ... aa: ADDRESS
 004         b b b b b b b b   bbbbbbbb: LENGTH
 005         0 0 0 0 0 0 F+L
```

SEEK (6 byte CDB)

```
 BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: SEEK
 000         Operation  code   CDB type: TYPE_009
 001         Unit# a a a a a   CDB data: 0B0000000000
 002         a a a a a a a a   Operation code = 0Bh
 003         a a a a a a a a   aa ... aa: ADDRESS
 004         0 0 0 0 0 0 0 0
 005         0 0 0 0 0 0 F+L
```

INQUIRY

```
 BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: INQUIRY
 000         Operation  code   CDB type: TYPE_003
 001         Unit# 0 0 0 0 0   CDB data: 120000000000
 002         0 0 0 0 a a a a   Operation code = 12h
 003         0 0 0 0 0 0 0 0   aaaa: FORMAT
 004         b b b b b b b b   bbbbbbbb: LENGTH
 005         0 0 0 0 0 0 F+L   DATA buffer: INQUIRED
```

MODE SELECT

```
 BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: MODE_SELECT
 000         Operation  code   CDB type: TYPE_010
 001         Unit# A 0 0 0 B   CDB data: 150000000000
 002         0 0 0 0 0 0 0 0   Operation code = 15h
 003         0 0 0 0 0 0 0 0   A: , PF
 004         c c c c c c c c   B: , SP
```

```
005          0 0 0 0 0 0 F+L   cccccccc: LENGTH
                               DATA buffer: PAGE_4
```

RESERVE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: RESERVE
000         Operation  code    CDB type: TYPE_011
001         Unit# A b b b C    CDB data: 160000000000
002         d d d d d d d d    Operation code = 16h
003         e e e e e e e e    A: , TRDPTY
004         e e e e e e e e    bbb: DEVICE
005         0 0 0 0 0 0 F+L    C: , EXTENT
                               ddddddddd: IDENTIFICATION
                               ee ... ee: LENGTH
                               DATA buffer: EXTENTS
```

RELEASE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: RELEASE
000         Operation  code    CDB type: TYPE_012
001         Unit# A b b b C    CDB data: 170000000000
002         d d d d d d d d    Operation code = 17h
003         0 0 0 0 0 0 0 0    A: , TRDPTY
004         0 0 0 0 0 0 0 0    bbb: DEVICE
005         0 0 0 0 0 0 F+L    C: , EXTENT
                               ddddddddd: IDENTIFICATION
```

COPY

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: COPY
000         Operation  code    CDB type: TYPE_004
001         Unit# 0 0 0 0 A    CDB data: 180000001000
002         b b b b b b b b    Operation code: 18h
003         b b b b b b b b    A: , PAD
004         b b b b b b b b    bb ... bb: LENGTH
005         0 0 0 0 0 0 F+L    DATA buffer: COPY_4
```

MODE SENSE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: MODE_SENSE
000         Operation  code    CDB type: TYPE_013
001         Unit# A 0 0 0 0    CDB data: 1A0004000000
002         B B c c c c c c    Operation code = 1Ah
003         0 0 0 0 0 0 0 0    A: , PF
004         d d d d d d d d    BB: CURRENT, CHANGEABLE, DEFAULT
005         0 0 0 0 0 0 F+L        SAVED
                               cccccc: PAGE_CODE
                               ddddddddd: LENGTH
                               DATA buffer: PAGE_4
```

TART/STOP UNIT

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: START/STOP_UNIT
000         Operation  code   CDB type: TYPE_014
001         Unit# 0 0 0 0 A   CDB data: 1B0000000000
002         0 0 0 0 0 0 0 0   Operation code = 1Bh
003         0 0 0 0 0 0 0 0   A: , IMMED
004         0 0 0 0 0 0 B C   B: , LOEJ
005         0 0 0 0 0 0 F+L   C: STOP, START
```

ECEIVE DIAGNOSTIC RESULTS

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: RECEIVE_DIAGNOSTIC_RESULTS
000         Operation  code   CDB type: TYPE_005
001         Unit# 0 0 0 0 0   CDB data: 1C0000000000
002         0 0 0 0 0 0 0 0   Operation code = 1Ch
003         a a a a a a a a   aa ... aa: LENGTH
004         a a a a a a a a
005         0 0 0 0 0 0 F+L
```

END DIAGNOSTIC

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: SEND_DIAGNOSTIC
000         Operation  code   CDB type: TYPE_006
001         Unit# 0 0 A B C   CDB data: 1D0000000000
002         0 0 0 0 0 0 0 0   Operation code = 1Dh
003         d d d d d d d d   A: , SELFTEST
004         d d d d d d d d   B: , DEVOFL
005         0 0 0 0 0 0 F+L   C: , UNITOFL
                              dddddddd: LENGTH
```

REVENT/ALLOW MEDIUM REMOVAL

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: PREVENT/ALLOW_MEDIUM_REMOV
000         Operation  code   CDB type: TYPE_015
001         Unit# 0 0 0 0 0   CDB data: 1E0000000000
002         0 0 0 0 0 0 0 0   Operation code = 1Eh
003         0 0 0 0 0 0 0 0   A: ALLOW, PREVENT
004         0 0 0 0 0 0 0 A
005         0 0 0 0 0 0 F+L
```

EAD CAPACITY

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: READ_CAPACITY
000         Operation  code   CDB type: TYPE_104
001         Unit# 0 0 0 0 A   CDB data: 25000000000000000000
002         b b b b b b b b   Operation code = 25h
003         b b b b b b b b   A: , RELADR
004         b b b b b b b b   bb ... bb: ADDRESS
005         b b b b b b b b   C: , PMI
```

```
 006          0 0 0 0 0 0 0 0   DATA buffer: CAPACITY
 007          0 0 0 0 0 0 0 0
 008          0 0 0 0 0 0 0 C
 009          0 0 0 0 0 0 F+L

READ (10 byte CDB)

 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: READ_EXTENDED
 000          Operation  code   CDB type: TYPE_105
 001          Unit# A B 0 0 C   CDB data: 28000000000000000000
 002          d d d d d d d d   Operation code = 28h
 003          d d d d d d d d   A: , DPO
 004          d d d d d d d d   B: , FUA
 005          d d d d d d d d   C: , RELADR
 006          0 0 0 0 0 0 0 0   dd ... dd: ADDRESS
 007          e e e e e e e e   ee ... ee: LENGTH
 008          e e e e e e e e
 009          0 0 0 0 0 0 F+L

WRITE (10 byte CDB)

 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: WRITE_EXTENDED
 000          Operation  code   CDB type: TYPE_105
 001          Unit# A B 0 0 C   CDB data: 2A000000000000000000
 002          d d d d d d d d   Operation code = 2Ah
 003          d d d d d d d d   A: , DPO
 004          d d d d d d d d   B: , FUA
 005          d d d d d d d d   C: , RELADR
 006          0 0 0 0 0 0 0 0   dd ... dd: ADDRESS
 007          e e e e e e e e   ee ... ee: LENGTH
 008          e e e e e e e e
 009          0 0 0 0 0 0 F+L

SEEK (10 byte CDB)

 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: SEEK_EXTENDED
 000          Operation  code   CDB type: TYPE_106
 001          Unit# 0 0 0 0 0   CDB data: 2B000000000000000000
 002          a a a a a a a a   Operation code = 2Bh
 003          a a a a a a a a   aa ... aa: ADDRESS
 004          a a a a a a a a
 005          a a a a a a a a
 006          0 0 0 0 0 0 0 0
 007          0 0 0 0 0 0 0 0
 008          0 0 0 0 0 0 0 0
 009          0 0 0 0 0 0 F+L

WRITE AND VERIFY

 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: WRITE_AND_VERIFY
```

```
000          Operation  code    CDB type: TYPE_107
001          Unit# A B C D E    CDB data: 2E000000000000000000
002          f f f f f f f f    Operation code = 2Eh
003          f f f f f f f f    A: , DPO
004          f f f f f f f f    B: , FUA
005          f f f f f f f f    C: , WRTSME
006          0 0 0 0 0 0 0 0    D: , BYTCHK
007          g g g g g g g g    E: , RELADR
008          g g g g g g g g    ff ... ff: ADDRESS
009          0 0 0 0 0 0 F+L    gg ... gg: LENGTH
```

VERIFY

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: VERIFY
000          Operation  code    CDB type: TYPE_108
001          Unit# A B 0 C D    CDB data: 2F000000000000000000
002          e e e e e e e e    Operation code = 2Fh
003          e e e e e e e e    A: , DPO
004          e e e e e e e e    B: , FUA
005          e e e e e e e e    C: , BYTCHK
006          0 0 0 0 0 0 0 0    D: , RELADR
007          f f f f f f f f    ee ... ee: ADDRESS
008          f f f f f f f f    ff ... ff: LENGTH
009          0 0 0 0 0 0 F+L
```

SEARCH DATA

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: SEARCH_DATA_HIGH
000          Operation  code    CDB type: TYPE_109
001          Unit# A 0 0 B C    CDB data: 30000000000000000000
002          d d d d d d d d    Operation code = 30h
003          d d d d d d d d    A: , INVERT
004          d d d d d d d d    B: , SPNDAT
005          d d d d d d d d    C: , RELADR
006          0 0 0 0 0 0 0 0    dd ... dd: ADDRESS
007          e e e e e e e e    ee ... ee: LENGTH
008          e e e e e e e e    DATA buffer: SEARCH_DATA
009          0 0 0 0 0 0 F+L

BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: SEARCH_DATA_EQUAL
000          Operation  code    CDB type: TYPE_109
001          Unit# A 0 0 B C    CDB data: 31000000000000000000
002          d d d d d d d d    Operation code = 31h
003          d d d d d d d d    A: , INVERT
004          d d d d d d d d    B: , SPNDAT
005          d d d d d d d d    C: , RELADR
006          0 0 0 0 0 0 0 0    dd ... dd: ADDRESS
007          e e e e e e e e    ee ... ee: LENGTH
008          e e e e e e e e    DATA buffer: SEARCH_DATA
009          0 0 0 0 0 0 F+L
```

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: SEARCH_DATA_LOW
000         Operation  code    CDB type: TYPE_109
001         Unit# A 0 0 B C    CDB data: 32000000000000000000
002         d d d d d d d d    Operation code = 32h
003         d d d d d d d d    A: , INVERT
004         d d d d d d d d    B: , SPNDAT
005         d d d d d d d d    C: , RELADR
006         0 0 0 0 0 0 0 0    dd ... dd: ADDRESS
007         e e e e e e e e    ee ... ee: LENGTH
008         e e e e e e e e    DATA buffer: SEARCH_DATA
009         0 0 0 0 0 0 F+L
```

SET LIMITS

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: SET_LIMITS
000         Operation  code    CDB type: TYPE_110
001         Unit# 0 0 0 A B    CDB data: 33000000000000000000
002         c c c c c c c c    Operation code = 33h
003         c c c c c c c c    A: , RDINH
004         c c c c c c c c    B: , WRINH
005         c c c c c c c c    cc ... cc: ADDRESS
006         0 0 0 0 0 0 0 0    dd ... dd: BLOCKS
007         d d d d d d d d
008         d d d d d d d d
009         0 0 0 0 0 0 F+L
```

PRE-FETCH

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: PREFETCH
000         Operation  code    CDB type: TYPE_111
001         Unit# 0 0 0 A B    CDB data: 34000000000000000000
002         c c c c c c c c    Operation code = 34h
003         c c c c c c c c    A: , IMMED
004         c c c c c c c c    B: , RELADR
005         c c c c c c c c    cc ... cc: ADDRESS
006         0 0 0 0 0 0 0 0    dd ... dd: LENGTH
007         d d d d d d d d
008         d d d d d d d d
009         0 0 0 0 0 0 F+L
```

FLUSH CACHE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: FLUSH_CACHE
000         Operation  code    CDB type: TYPE_112
001         Unit# 0 0 0 0 A    CDB data: 35000000000000000000
002         b b b b b b b b    Operation code = 35h
003         b b b b b b b b    A: , RELADR
004         b b b b b b b b    bb ... bb: ADDRESS
005         b b b b b b b b    cc ... cc: LENGTH
```

```
006          0 0 0 0 0 0 0 0
007          c c c c c c c c
008          c c c c c c c c
009          0 0 0 0 0 0 F+L
```

LOCK/UNLOCK CACHE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: LOCK/UNLOCK_CACHE
000          Operation  code   CDB type: TYPE_111
001          Unit# 0 0 0 A B   CDB data: 360000000000000000000
002          c c c c c c c c   Operation code = 36h
003          c c c c c c c c   A: , IMMED
004          c c c c c c c c   B: , RELADR
005          c c c c c c c c   cc ... cc: ADDRESS
006          0 0 0 0 0 0 0 0   dd ... dd: LENGTH
007          d d d d d d d d
008          d d d d d d d d
009          0 0 0 0 0 0 F+L
```

READ DEFECT DATA

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: READ_DEFECT_DATA
000          Operation  code   CDB type: TYPE_113
001          Unit# 0 0 0 0 0   CDB data: 3700000000000000000000
002          0 0 0 A B c c c   Operation code = 37h
003          0 0 0 0 0 0 0 0   A: , P
004          0 0 0 0 0 0 0 0   B: , G
005          0 0 0 0 0 0 0 0   ccc: FORMAT
006          0 0 0 0 0 0 0 0   dd ... dd: LENGTH
007          d d d d d d d d
008          d d d d d d d d
009          0 0 0 0 0 0 F+L
```

COMPARE

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: COMPARE
000          Operation  code   CDB type: TYPE_101
001          Unit# 0 0 0 0 A   CDB data: 390000000000000000000
002          0 0 0 0 0 0 0 0   Operation code = 39h
003          b b b b b b b b   A; , PAD
004          b b b b b b b b   bb ... bb: LENGTH
005          b b b b b b b b
006          0 0 0 0 0 0 0 0
007          0 0 0 0 0 0 0 0
008          0 0 0 0 0 0 0 0
009          0 0 0 0 0 0 F+L
```

COPY AND VERIFY

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: COPY_AND_VERIFY
```

```
 000          Operation  code    CDB type: TYPE_102
 001          Unit# 0 0 0 A B    CDB data: 3A000000000000000000
 002          0 0 0 0 0 0 0 0    Operation code = 3Ah
 003          c c c c c c c c    A: , BYTCHK
 004          c c c c c c c c    B: , PAD
 005          c c c c c c c c    cc ... cc: LENGTH
 006          0 0 0 0 0 0 0 0
 007          0 0 0 0 0 0 0 0
 008          0 0 0 0 0 0 0 0
 009          0 0 0 0 0 0 F+L
```

WRITE BUFFER

```
 BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: WRITE_BUFFER
 000          Operation  code    CDB type: TYPE_103
 001          Unit# 0 0 0 a a    CDB data: 3B000000000000000000
 002          b b b b b b b b    Operation code = 3Bh
 003          c c c c c c c c    aa: MODE
 004          c c c c c c c c    bbbbbbbb: BUFFER_ID
 005          c c c c c c c c    cc ... cc: BUFFER_OFFSET
 006          d d d d d d d d    dd ... dd: LENGTH
 007          d d d d d d d d
 008          d d d d d d d d
 009          0 0 0 0 0 0 F+L
```

READ BUFFER

```
 BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: READ_BUFFER
 000          Operation  code    CDB type: TYPE_103
 001          Unit# 0 0 0 a a    CDB data: 3C000000000000000000
 002          b b b b b b b b    Operation code = 3Ch
 003          c c c c c c c c    aa: MODE
 004          c c c c c c c c    bbbbbbbb: BUFFER_ID
 005          c c c c c c c c    cc ... cc: BUFFER_OFFSET
 006          d d d d d d d d    dd ... dd: LENGTH
 007          d d d d d d d d
 008          d d d d d d d d
 009          0 0 0 0 0 0 F+L
```

READ LONG

```
 BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: READ_LONG
 000          Operation  code    CDB type: TYPE_114
 001          Unit# 0 0 0 A B    CDB data: 3E000000000000000000
 002          c c c c c c c c    Operation code = 3Eh
 003          c c c c c c c c    A: , CORRCT
 004          c c c c c c c c    B: , RELADR
 005          c c c c c c c c    cc ... cc: ADDRESS
 006          0 0 0 0 0 0 0 0    dd ... dd: LENGTH
 007          d d d d d d d d
```

```
008          d d d d d d d d
009          0 0 0 0 0 0 F+L
```

WRITE LONG

```
BYTE -- bit 7-6-5-4-3-2-1-0    CDB name: WRITE_LONG
000          Operation  code   CDB type: TYPE_112
001          Unit# 0 0 0 0 A   CDB data: 3F000000000000000000
002          b b b b b b b b   Operation code = 3Fh
003          b b b b b b b b   A: , RELADR
004          b b b b b b b b   bb ... bb: ADDRESS
005          b b b b b b b b   cc ... cc: LENGTH
006          0 0 0 0 0 0 0 0
007          c c c c c c c c
008          c c c c c c c c
009          0 0 0 0 0 0 F+L
```

APPENDIX C - SCSI-2 SEQUENTIAL ACCESS DDL

INTRODUCTION

This appendix is a catalog of the commands defined in the Device Description Library provided on the program disk as file SCSI2_SA.DDL. This DDL file is based on a preliminary draft of the proposed SCSI-2 standard dated October 31, 1986. This rendition of the SCSI-2 sequential access DDL must be considered only a prototype which requires editing by you to match the specific implementation of your particular target. To do this you will need to refer to the SCSI manual provided by the target vendor. Presently the proposed standard is in considerable flux and is undergoing changes by, and at the sole discretion of, the ANSI X3T9.2 Task Group.

Table C-1 provides an alphabetical list of the commands in the catalog. Each catalog entry includes the template of the CDB, the command name, the referenced TYPE definition, the CDB data as a string of hex digits, the name of each defined field in the CDB and the DATA buffer attached to the command (refer to the example COPY command below).

Each bit of the CDB template is coded. "Operation code" identifies the first byte of each CDB. Its assigned value is to the right of the template. The target LUN address is inserted in the "Unit#" field when the command is issued to a target. A string of a lower case letter identifies a named field. The name assigned each field can be found to the right of the template. A string of an upper case letter identifies an enumerated field. The name of each enumeration is listed to the right of the letter in ascending order, separated by comas. Unnamed enumerations appear as blanks. For example, the enumerated value 1 of field A of the COPY CDB has been assigned the label PAD while the value 0 has been assigned no label. Bit positions not assigned to a field are identified in the template by 0.

If there is a DATA buffer assigned to a CDB other than the normal default buffer .SMALL, it is identified to the right of the template.

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: COPY
000         Operation  code   CDB type: TYPE_004
001         Unit# 0 0 0 0 A   CDB data: 180000001000
002         b b b b b b b b   Operation code = 18h
003         b b b b b b b b   A: , PAD
004         b b b b b b b b   bb ... bb: LENGTH
005         0 0 0 0 0 0 F+L   DATA buffer: COPY_4
```

As mentioned above, you will need to edit this DDL to match the implementation of your particular target. We suggest you save any changes under a file name different than SCSI2_SA. Below are some additional hints (cans and can'ts) for you to consider as you edit the DDL.

First, you CAN:

create a new FORM or TYPE field by assigning bits presently unassigned to the new field,

change the name of a TYPE or FORM field,

change the name of a CDB or DATA buffer,

change the DATA buffer attached to a CDB.

Second, you CAN, BUT not without impact elsewhere:

change the size or kind of a field. If a resized field encroachs into another defined field, that other field is deleted.

change the size of a TYPE template. If you do you must delete and recreate all CDB's referencing the changed template.

change the size of a FORM template. If you do you must delete and recreate all DATA buffers referencing the changed template.

rename or delete a TYPE or FORM template. If you do you must delete and recreate all CDB's or DATA buffers referencing the renamed or deleted template.

delete a TYPE or FORM field. If you do all subsequent fields in the TYPE or FORM will be deleted.

Lastly, you CAN'T:

change the TYPE of a CDB (You must delete the existing CDB and recreate it with the new TYPE assigned),

change the FORM of a DATA buffer (You must delete the existing DATA and recreate is with the new FORM assigned).

Long names for CDB's, DATA buffers and FORM and TYPE fields should be avoided. They do impact program performance,

increase DDL size and require more typing at the keyboard. For understandability, the names used in this DDL are particularly verbose. You may wish to edit these names to something equally meaningful to you while reducing their length.

Keep in mind that any edits you make on a TYPE or FORM template (other than field name changes) may adversely impact the CDB's and DATA buffers referencing those templates.

Table B-1 Commands for Sequential Assess Devices

TEST UNIT READY

BYTE -- bit 7-6-5-4-3-2-1-0	CDB name: TEST_UNIT_READY
000 Operation code	CDB type: TYPE_001
001 Unit# 0 0 0 0 0	CDB data: 000000000000
002 0 0 0 0 0 0 0 0	Operation code = 00h
003 0 0 0 0 0 0 0 0	
004 0 0 0 0 0 0 0 0	
005 0 0 0 0 0 0 F+L	

REWIND

BYTE -- bit 7-6-5-4-3-2-1-0	CDB name: REWIND
000 Operation code	CDB type: TYPE_001
001 Unit# 0 0 0 0 0	CDB data: 010000000000
002 0 0 0 0 0 0 0 0	Operation code = 01h
003 0 0 0 0 0 0 0 0	
004 0 0 0 0 0 0 0 0	
005 0 0 0 0 0 0 F+L	

REQUEST SENSE

BYTE -- bit 7-6-5-4-3-2-1-0	CDB name: REQUEST_SENSE
000 Operation code	CDB type: TYPE_002
001 Unit# 0 0 0 0 0	CDB data: 030000000000
002 0 0 0 0 0 0 0 0	Operation code = 03h
003 0 0 0 0 0 0 0 0	aaaaaaaa: LENGTH
004 a a a a a a a a	DATA buffer: SENSED
005 0 0 0 0 0 0 F+L	

READ BLOCK LIMITS

BYTE -- bit 7-6-5-4-3-2-1-0	CDB name: READ_BLOCK_LIMITS
000 Operation code	CDB type: TYPE_001
001 Unit# 0 0 0 0 0	CDB data: 050000000000
002 0 0 0 0 0 0 0 0	Operation code = 05h
003 0 0 0 0 0 0 0 0	
004 0 0 0 0 0 0 0 0	
005 0 0 0 0 0 0 F+L	

READ

BYTE -- bit 7-6-5-4-3-2-1-0	CDB name: READ
000 Operation code	CDB type: TYPE_008
001 Unit# 0 0 0 A B	CDB data: 080000000000
002 c c c c c c c c	Operation code = 08h
003 c c c c c c c c	A: , SILI
004 c c c c c c c c	B: , FIXED
005 0 0 0 0 0 0 F+L	cc ... cc: LENGTH

WRITE

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: WRITE
000         Operation  code   CDB type: TYPE_009
001         Unit# 0 0 0 0 A   CDB data: 090000000000
002         b b b b b b b b   Operation code = 09h
003         b b b b b b b b   A: , FIXED
004         b b b b b b b b   bb ... bb: LENGTH
005         0 0 0 0 0 0 F+L
```

TRACK SELECT

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: TRACK_SELECT
000         Operation  code   CDB type: TYPE_010
001         Unit# 0 0 0 0 0   CDB data: 0B0000000000
002         0 0 0 0 0 0 0 0   Operation code = 0Bh
003         0 0 0 0 0 0 0 0   aaaaaaaa: TRACK
004         a a a a a a a a
005         0 0 0 0 0 0 F+L
```

READ REVERSE

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: READ_REVERSE
000         Operation  code   CDB type: TYPE_008
001         Unit# 0 0 0 A B   CDB data: 0F0000000000
002         c c c c c c c c   Operation code = 0Fh
003         c c c c c c c c   A: , SILI
004         c c c c c c c c   B: , FIXED
005         0 0 0 0 0 0 F+L   cc ... cc: LENGTH
```

WRITE FILEMARKS

```
BYTE -- bit 7-6-5-4-3 2 1-0   CDB name: WRITE_FILEMARKS
000         Operation  code   CDB type: TYPE_011
001         Unit# 0 0 0 0 A   CDB data: 100000000000
002         b b b b b b b b   Operation code = 10h
003         b b b b b b b b   A: , IMMED
004         b b b b b b b b   bb ... bb: FILEMARKS
005         0 0 0 0 0 0 F+L
```

SPACE

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: SPACE
000         Operation  code   CDB type: TYPE_012
001         Unit# 0 0 0 A A   CDB data: 110000000000
002         b b b b b b b b   Operation code = 11h
003         b b b b b b b b   AA: BLOCKS, FILEMARKS
004         b b b b b b b b       SEQUENTIAL, END-OF-DATA
005         0 0 0 0 0 0 F+L   bb ... bb: COUNT
```

INQUIRY

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: INQUIRY
000	Operation code	CDB type: TYPE_003
001	Unit# 0 0 0 0 0	CDB data: 120000000000
002	0 0 0 0 a a a a	Operation code = 12h
003	0 0 0 0 0 0 0 0	aaaa: FORMAT
004	b b b b b b b b	bbbbbbbb: LENGTH
005	0 0 0 0 0 0 F+L	DATA buffer: INQUIRED

VERIFY

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: VERIFY
000	Operation code	CDB type: TYPE_013
001	Unit# 0 0 A B C	CDB data: 130000000000
002	d d d d d d d d	Operation code = 13h
003	d d d d d d d d	A: IMMED
004	d d d d d d d d	B: BYTCMP
005	0 0 0 0 0 0 F+L	C: FIXED
		dd ... dd: LENGTH

RECOVER BUFFERED DATA

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: RECOVER_BUFFERED_DATA
000	Operation code	CDB type: TYPE_009
001	Unit# 0 0 0 0 A	CDB data: 140000000000
002	b b b b b b b b	Operation code = 14h
003	b b b b b b b b	A: , FIXED
004	b b b b b b b b	bb ... bb: LENGTH
005	0 0 0 0 0 0 F+L	

MODE SELECT

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: MODE_SELECT
000	Operation code	CDB type: TYPE_014
001	Unit# A 0 0 0 B	CDB data: 150000000000
002	0 0 0 0 0 0 0 0	Operation code = 15h
003	0 0 0 0 0 0 0 0	A: , PF
004	c c c c c c c c	B: , SP
005	0 0 0 0 0 0 F+L	cc ... cc: LENGTH
		DATA buffer: PAGE_0

RESERVE UNIT

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: RESERVE_UNIT
000	Operation code	CDB type: TYPE_015
001	Unit# A b b b 0	CDB data: 160000000000
002	0 0 0 0 0 0 0 0	Operation code = 16h
003	0 0 0 0 0 0 0 0	A: , THIRDPTY
004	0 0 0 0 0 0 0 0	bbb: THIRDPTY_ID

```
 005          0 0 0 0 0 0 F+L
```

RELEASE UNIT

```
 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: RELEASE_UNIT
 000          Operation  code   CDB type: TYPE_015
 001          Unit# A b b b 0   CDB data: 170000000000
 002          0 0 0 0 0 0 0 0   Operation code = 17h
 003          0 0 0 0 0 0 0 0   A: , THIRDPTY
 004          0 0 0 0 0 0 0 0   bbb: THIRDPTY_ID
 005          0 0 0 0 0 0 F+L
```

COPY

```
 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: COPY
 000          Operation  code   CDB type: TYPE_004
 001          Unit# 0 0 0 0 A   CDB data: 180000001000
 002          b b b b b b b b   Operation code = 18h
 003          b b b b b b b b   A: , PAD
 004          b b b b b b b b   bb ... bb: LENGTH
 005          0 0 0 0 0 0 F+L   DATA buffer: COPY_4
```

ERASE

```
 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: ERASE
 000          Operation  code   CDB type: TYPE_016
 001          Unit# 0 0 0 A B   CDB data: 190000000000
 002          0 0 0 0 0 0 0 0   Operation code = 19h
 003          0 0 0 0 0 0 0 0   A: , IMMED
 004          0 0 0 0 0 0 0 0   B: , LONG
 005          0 0 0 0 0 0 F+L
```

MODE SENSE

```
 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: MODE_SENSE
 000          Operation  code   CDB type: TYPE_017
 001          Unit# 0 0 0 0 0   CDB data: 1A00000000000
 002          A A b b b b b b   Operation code = 1Ah
 003          0 0 0 0 0 0 0 0   AA: CURRENT, CHANGEABLE
 004          c c c c c c c c       DEFAULT, SAVED
 005          0 0 0 0 0 0 F+L   bbbbbb: PAGE_CODE
                                cccccccc: LENGTH
                                DATA buffer: PAGE_0
```

LOAD/UNLOAD

```
 BYTE --- bit 7-6-5-4-3-2-1-0   CDB name: LOAD/UNLOAD
 000          Operation  code   CDB type: TYPE_018
 001          Unit# 0 0 0 0 A   CDB data: 1B00000000000
 002          0 0 0 0 0 0 0 0   Operation code = 1Bh
```

```
003          0 0 0 0 0 0 0 0   A: , IMMED
004          0 0 0 0 0 0 B C   B: , RETEN
005          0 0 0 0 0 0 F+L   C: UNLOAD, LOAD
```

RECEIVE DIAGNOSTIC RESULTS

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: RECEIVE_DIAGNOSTIC_RESULTS
000          Operation  code  CDB type: TYPE_005
001          Unit# 0 0 0 0 0  CDB data: 1C0000000000
002          0 0 0 0 0 0 0 0  Operation code = 1Ch
003          a a a a a a a a  aa ... aa: LENGTH
004          a a a a a a a a
005          0 0 0 0 0 0 F+L
```

SEND DIAGNOSTIC

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: SEND_DIAGNOSTIC
000          Operation  code  CDB type: TYPE_006
001          Unit# 0 0 A B C  CDB data: 1D0000000000
002          0 0 0 0 0 0 0 0  Operation  code = 1Dh
003          d d d d d d d d  A: , SELFTEST
004          d d d d d d d d  B: , DEVOFL
005          0 0 0 0 0 0 F+L  C: , UNITOFL
                              dd ... dd: LENGTH
```

PREVENT/ALLOW MEDIUM REMOVAL

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: PREVENT/ALLOW_MEDIUM_REMOV
000          Operation  code  CDB type: TYPE_019
001          Unit# 0 0 0 0 0  CDB data: 1E0000000000
002          0 0 0 0 0 0 0 0  Operation code = 1Eh
003          0 0 0 0 0 0 0 0  A: ALLOW, PREVENT
004          0 0 0 0 0 0 0 A
005          0 0 0 0 0 0 F+L
```

READ LOG

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: READ_LOG
000          Operation  code  CDB type: TYPE_020
001          Unit# 0 0 0 0 A  CDB data: 1F0000000000
002          0 0 0 0 0 0 0 0  Operation code = 1Fh
003          b b b b b b b b  A: , NLR
004          b b b b b b b b  bb ... bb: LENGTH
005          0 0 0 0 0 0 F+L
```

LOCATE

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: LOCATE
000          Operation  code  CDB type: TYPE_104
001          Unit# 0 0 A B C  CDB data: 2B000000000000000000
```

```
002          0 0 0 0 0 0 0 0   Operation code = 2Bh
003          d d d d d d d d   A: , BT
004          d d d d d d d d   B: , CP
005          d d d d d d d d   C: , IMMED
006          d d d d d d d d   dd ... dd: ADDRESS
007          0 0 0 0 0 0 0 0   eeeeeeee: LENGTH
008          e e e e e e e e
009          0 0 0 0 0 0 F+L
```

READ POSITION

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: READ_POSITION
000          Operation  code   CDB type: TYPE_105
001          Unit# 0 0 0 0 0   CDB data: 34000000000000000000
002          0 0 0 0 0 0 0 0   Operation code = 34h
003          0 0 0 0 0 0 0 0   DATA buffer: POSITION
004          0 0 0 0 0 0 0 0
005          0 0 0 0 0 0 0 0
006          0 0 0 0 0 0 0 0
007          0 0 0 0 0 0 0 0
008          0 0 0 0 0 0 0 0
009          0 0 0 0 0 0 F+L
```

COMPARE

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: COMPARE
000          Operation  code   CDB type: TYPE_101
001          Unit# 0 0 0 0 A   CDB data: 39000000000000000000
002          0 0 0 0 0 0 0 0   Operation code = 39h
003          b b b b b b b b   A: , PAD
004          b b b b b b b b   bb ... bb: LENGTH
005          b b b b b b b b
006          0 0 0 0 0 0 0 0
007          0 0 0 0 0 0 0 0
008          0 0 0 0 0 0 0 0
009          0 0 0 0 0 0 F+L
```

COPY AND VERIFY

```
BYTE -- bit 7-6-5-4-3-2-1-0   CDB name: COPY_AND_VERIFY
000          Operation  code   CDB type: TYPE_102
001          Unit# 0 0 0 A B   CDB data: 3A000000000000000000
002          0 0 0 0 0 0 0 0   Operation code = 3Ah
003          c c c c c c c c   A: , BYTCHK
004          c c c c c c c c   B: , PAD
005          c c c c c c c c   cc ... cc: LENGTH
006          0 0 0 0 0 0 0 0
007          0 0 0 0 0 0 0 0
008          0 0 0 0 0 0 0 0
009          0 0 0 0 0 0 F+L
```

WRITE BUFFER

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: WRITE_BUFFER
000	Operation code	CDB type: TYPE_103
001	Unit# 0 0 0 a a	CDB data: 3B000000000000000000
002	b b b b b b b b	Operation code = 3Bh
003	c c c c c c c c	aa: MODE
004	c c c c c c c c	bbbbbbbb: BUFFER_ID
005	c c c c c c c c	cc ... cc: BUFFER_OFFSET
006	d d d d d d d d	dd ... dd: LENGTH
007	d d d d d d d d	
008	d d d d d d d d	
009	0 0 0 0 0 0 F+L	

READ BUFFER

BYTE -- bit 7-6-5-4-3-2-1-0		CDB name: READ_BUFFER
000	Operation code	CDB type: TYPE_103
001	Unit# 0 0 0 a a	CDB data: 3C000000000000000000
002	b b b b b b b b	Operation code = 3Ch
003	c c c c c c c c	aa: MODE
004	c c c c c c c c	bbbbbbbb: BUFFER_ID
005	c c c c c c c c	cc ... cc: BUFFER_OFFSET
006	d d d d d d d d	dd ... dd: LENGTH
007	d d d d d d d d	
008	d d d d d d d d	
009	0 0 0 0 0 0 F+L	