pmEasy®

Protected Mode Environment

Developer's Guide

Version 2.2
March 2010

by
David Moore

© Copyright 1994-2010

Micro Digital Associates, Inc.
2900 Bristol Street #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

All rights reserved.

Revisions

date	ver	comments
1/95	1.0	first release
8/95	1.x	corrections to Soft-Scope appendix, mostly
2/00	2.0	thorough revision and documentation of new features
10/03	2.2	update, corrections, and reordering of chapters
3/10	2.2	update

pmEasy is a Trademark of Micro Digital, Inc.
smx is a Registered Trademark of Micro Digital, Inc.

Contents

Installation **1**
Installation 1
Directory Structure 1
Introduction **3**
Summary of Features 3
What pmEasy Does 3
pmEasy16 vs. pmEasy32 4
Building pmEasy 4
Tools **7**
Tools for Building the Application 7
Tools for Building pmEasy 7
Debuggers 7
Running pmEasy **9**
Running pmEasy from DOS 9
Bootloading pmEasy from Disk with BIOS (no DOS) 14
Running pmEasy from ROM 16
Restarting pmEasy 16
Preparing for Release 17
Loading the Application **19**
Loading the Application from Disk 19
Loading from Other Sources with DISKRD 21
Exe Loaders 22
Hex Loader 25
Application in ROM 26
Application Development **27**
Where Do I Start? 27
Building the Sample Applications 27
Running the Sample Applications 28
Your Application 28
Module Inclusion Order 29
Defines Used in Makefiles for MINAPPs 29
Startup Code and Entry Point 30
DOS Emulation 31
Win32 Emulation 31
Writing ISRs 31
Saving and Restoring Interrupt Vectors 34
Uninitialized Static Variables in C 34
Flat Mode Segments for pmEasy32 34

Contents

Tiled Descriptors for pmEasy16 35

Debugging the Application 37

Debuggers Supported 37
Soft-Scope 37
Techniques If Not Using a Debugger 42

pmEasy Operation Summary 45

Entering Protected Mode 45
Running in Protected Mode 46
Loading, Running, and Exiting the Application 47
Exiting pmEasy 48

pmEasy Internals 51

Heap Structure and Initialization 51
Buffer Area 53
PIC Configuration 53
Small Model Only 54
SS == DS 54
Constructing Far Pointers 54

Altering pmEasy 55

Far Addresses 55
Write pmEasy ISRs in Assembly or with Shells 56

Debugging pmEasy 57

Real Mode Code 57
Protected Mode Code 57
Markers 58

Configuration Reference 59

Alphabetical Cross Reference of Configuration Settings 59
Configuration Settings Reference 60

DPMI API Reference 69

DPMI Assembly Interface 70
Calling from C 71
C API Summary 71
DPMI / CPMI Reference 72

Error Reference 85

Return Code Reference 95

dpmiInit Return Codes 95
Loader Return Codes 95

Tips 97

Troubleshooting 99

Appendix A: File Formats 103

New Executable File Format (NE) 103
Portable Executable File Format (PE) 104

Appendix B: Boot Sector 107
Boot Sector Layout 107
Restoring the Boot Sector (MBR) 107
Appendix C: DiskOnChip® 109
Index 111

Installation

Installation

Run the installation utility on the media supplied. If pmEasy is purchased with the SMX® RTOS, it is part of that file set and installed in the PME subdirectory of the directory where SMX is installed.

Directory Structure

All pmEasy files are put under the directory you specified during install, or PME if installed with SMX. Summary of the organization of files (not all directories are shown):

Directory	Contents
PME	pmEasy main source files
CSIMON	Soft-Scope debug monitor
LOAD	Loader
DISKRD	Simple disk reader
DISKRD21	DOS int 21h disk reader
MINAAPP/MINCAPP	Minimal assembly/C application
MC.P3, BC.P3, etc.	Build directory
LR, SR, etc.	Object/EXE directory
MISC	Binding and bootloading support
PME16 or PME32	pmEasy build directory
REL	Object/EXE directory
DBG	Object/EXE directory

PROTOPM (if SMX is purchased)

PME contains the source files for pmEasy.

PME16 or **PME32** contains the makefiles to build pmEasy. (Makefiles are for Microsoft 16-bit C and MASM.) The **REL** and **DBG** subdirectories hold the pmEasy executables, object files, and map files for Release and Debug builds, respectively. The Debug build is for debugging the application, not pmEasy.

The **CSIMON** directory is shipped only if you purchase the Soft-Scope® debugger. This directory contains the remote monitor for the Soft-Scope debugger. It has been modified for use with pmEasy. See the chapter *Debugging the Application* for more information about Soft-Scope.

The **LOAD** directory contains the application loader and disk readers.

MINAAPP and **MINCAPP** are minimal assembly and C applications, respectively, which can be loaded under pmEasy. The Protosystem (see PROTOPM below) is a more advanced

application, supplied with our real-time multitasking kernel, smx. If you are using smx, you should focus on the Protosystem and ignore these simple applications. Otherwise, these are intended as the platforms for developing your application. This is discussed in the *Application Development* chapter of this manual.

The **MISC** directory contains various subdirectories and files, including utilities and directions for making a bootloadable disk or diskette. Also files for binding pmEasy to the application are located here.

PROTOPM is part of an smx installation. It is a skeletal smx application intended to be the basis for your smx application. See the smx Quick Start manual for more discussion.

Introduction

x86 processors, starting with the 286[1], have a second mode of operation, called *protected mode*. They start in real mode and require software to switch them into protected mode. This is one of the primary services pmEasy performs.

Summary of Features

- Protected Mode Entry and Exit
  - Initializes descriptor tables and processor registers.
  - Configures Programmable Interrupt Controllers (PICs).
  - Implements and hooks processor fault handlers
  - Returns to DOS if started from DOS.
- Application Load from Disk in EXE Format (NE or PE)
  - Avoids use of linker/locator. Compiler's linker used instead.
  - Disk readers for embedded target and DOS target.
  - Various media supported. See list in *Loading the Application* chapter.
  - Can be bound to application to form single EXE.
- DPMI 1.0 Server (subset)
  - Memory Management
  - LDT Descriptor Management
  - Interrupt Management
  - DPMI 300h for initializing special devices with real mode driver
  - C API for use from application
- Execution from Disk Boot Sector or DOS Prompt
- Debugger Support
- Framework for Writing DOS int 21h Emulation Routines
- Sample Applications to Start From
- Simple Video Routines

What pmEasy Does

pmEasy provides a simple protected mode environment under which an x86 protected mode application can run. It is a dynamic environment, separate from the application. It sets up descriptor tables and switches into protected mode. Then it loads the application from disk[2] and starts it running. It provides DPMI services to the application for memory management, interrupt management, and descriptor table management. Diagnostic trap handlers are installed to provide information about processor faults such as GPFs. These

---

[1] pmEasy requires a 386 or higher, since it makes extensive use of 32-bit registers. A 286 version is possible but has never been implemented due to lack of demand.
[2] See the list of media supported in chapter *Loading the Application*.

report the selector and offset of the instruction that caused the fault. It can be bootloaded from disk boot sector or run from the DOS prompt, and it will exit cleanly to DOS. Additionally, it can be bound to the application, in which case it appears DOS is directly hosting it. It can also be located and run from ROM.

pmEasy is an alternative to complex environments such as Windows™ and DOS extenders. Unlike these, pmEasy is simple because it remains in protected mode, so it avoids the complexities of repeatedly switching to real mode or virtual 86 mode which complicate DOS extenders. Furthermore, unlike these alternatives, source code is provided so you can see what pmEasy does and alter it if desired. The last part of this manual discusses configuration and modification of pmEasy, including tips for debugging it.

pmEasy is also an alternative to using a linker/locator. The standard linker supplied with the Borland, Microsoft, or Paradigm compiler is used to produce an EXE file, which is relocatable and runs from RAM. There are no worries about setting location addresses and avoiding overlaps, copying initialized data to RAM, creation of descriptor tables, or complicated startup code. It is common for x86 embedded systems to have disks or flash disks, making pmEasy a good option for many applications. Also note also that the handful of disk routines needed by the loader could be replaced with code that loads the file from any source, such as linear flash, ROM, or a data link. (This would have to be implemented by the user.)

pmEasy can load the application from many types of disks. See the *Media Supported* section in the chapter *Loading the Application* for a list.

pmEasy is integrated with the Soft-Scope® debugger to provide full source-level debugging of the application. See the chapter *Debugging the Application* for more information about Soft-Scope.

pmEasy is integrated with the SMX® RTOS, which is a great benefit if your project has a multitasking requirement. For those not using SMX, minimal sample applications are provided to begin development from. MINAAPP and MINCAPP show how little is required by pmEasy to write an application.

pmEasy16 vs. pmEasy32

pmEasy16 supports 16-bit protected mode applications, and **pmEasy32** supports 32-bit applications. Both are covered in this manual. Where reference is made to pmEasy, the discussion applies to both versions. The primary differences are the application loaders and DPMI servers. 16- and 32-bit applications have different file formats and are produced using different compilers and linkers. Each version of pmEasy has its own application loader to load the appropriate file format: New Executable (NE) for 16-bit applications and Portable Executable (PE) for 32-bit applications. pmEasy16 has a 16-bit DPMI server while pmEasy32 has a 32-bit DPMI server. This is necessary to handle 16-bit offsets vs. 32-bit offsets.

Building pmEasy

pmEasy provides numerous configuration options and is supplied with full source code, so it is possible to change it to suit your needs. After making any changes, pmEasy must be rebuilt. Both pmEasy16 and pmEasy32 are built with MASM v6.11 or higher and the

Microsoft Visual C++ 16-bit compiler and linker. See the *Tools* chapter for more information.

1. Change directory to PME\PME16 or PME\PME32.

2. Type one of these commands:

C:>**mak r**	to build the Release version of pmEasy
C:>**mak d**	to build the Debug version of pmEasy

The Debug version of pmEasy is used for debugging the application.

If you have problems building pmEasy, please see the Troubleshooting section of this manual.

We suggest that you initially use the pre-built pmEasy executables provided and start developing the application before changing pmEasy configuration settings or modifying source code. The *Configuration Reference*, later in this manual, lists all of the configuration options. Other chapters discuss how pmEasy works and other issues related to modifying it.

Tools

Tools for Building the Application

16-bit: Borland, Microsoft, or Paradigm 16-bit compiler. (NE)

32-bit: Borland, Microsoft, or Paradigm 32-bit compiler. (PE)

It is likely that other compilers that produce NE or PE files can be used to build the application, but these are the only compilers we use.

Tools for Building pmEasy

pmEasy is built with these tools:

Microsoft Visual C++ v1.52 (16-bit compiler and linker)
Microsoft Assembler (MASM) v6.11 or later

Note that **both** pmEasy16 and pmEasy32 are built with the **16-bit tools** because they are both linked as 16-bit applications. This follows from the requirement that they run from DOS or are bootloaded at power up. pmEasy32 users, please note that the *application* is built with 32-bit tools. Do not worry about use of these old tools for building pmEasy, since it is seldom rebuilt. What is important is the tools you use for application development.

These tools were discontinued by Microsoft long ago but have been available in various forms over the years, such as bundled with books. Please consult the release notes, PME22.TXT, for suggestions about where to obtain them. Contact Micro Digital if you need further assistance.

If you have problems building pmEasy, please see the Troubleshooting section of this manual.

Debuggers

Soft-Scope®: This is currently the only debugger supported for debugging EXE files. Modifications to the target monitor were necessary so it could be linked to pmEasy. This modified monitor is supplied with pmEasy in the CSIMON directory. CSi-Locate is used to extract the debug symbolics from the .exe and put them in a form readable by Soft-Scope.

An alternative is to use Paradigm C++ and debug a located application and then build as an EXE for release.

See the chapter *Debugging the Application* later in this manual for more discussion of debugging.

Running pmEasy

If pmEasy hangs or has other run-time problems, please see the Troubleshooting chapter and Error Reference for guidance.

Running pmEasy from DOS

DOS Command Prompt

pmEasy can be run from the DOS command prompt, just like any DOS executable. This is convenient during development. For example:

C:> **pme32**

By default, it loads A:\APP.EXE, or whatever program the **appName** variable specifies, in PME.ASM. The name of the program to load can also be specified on the command line (if PARSECMDLINE == 1). For example:

C:> **pme32 c:\app**

Another option is to bind pmEasy to the application and run the application from the command line. For example:

C:> **app**

These options are discussed further in the subsections that follow.

pmEasy's loader displays information to the screen about segments or sections that it is loading. When the load is complete pmEasy waits for the Enter key to be pressed and then starts the application running. When the application finishes, it returns to pmEasy, and pmEasy displays the application return code, in the AX register.

All screen display and keyboard waits can be disabled easily by setting the VIDEO and KEYBOARD options to 0 in pme.inc and pme.h. Also, multiple versions of the pre-built loader are supplied, with varying levels of screen display (including none), in case you wish to leave VIDEO == 1 to display other output. If the keyboard is disabled, pmEasy immediately starts the application when the load completes and immediately returns to DOS or reboots when the loaded program exits. For more information about configuration settings, see the chapter *Configuration Reference* later in this manual. Also, source code is supplied, allowing any additional modification to pmEasy's behavior.[3]

Windows 9x MS-DOS Command Prompt

pmEasy will not run from the MS-DOS Command Prompt in Windows 9x. You must boot to MS-DOS mode from the menu that appears when you press F8 at startup or, if already

[3] Disk driver source code is available at additional cost.

running Windows, select Shutdown, then Restart the computer in MS-DOS mode. This is because pmEasy will not run under another protected mode environment. Alternatively, boot to a DOS system floppy.

Windows NT does not offer an option to boot to DOS, so boot to a DOS floppy to run pmEasy.

We recommend running from a computer other than your development system. It is most convenient if the testing system has DOS installed on the hard disk, since booting to a floppy is slow.

Memory Managers

Memory managers such as EMM386 and 386MAX conflict with pmEasy and prevent it from successfully entering protected mode. If one is present, pmEasy will abort. pmEasy checks for the presence of a DPMI server and aborts with the message “ABORTED. DPMI SERVER ALREADY PRESENT” if one is present. Note that only DPMI servers are detected by this test, so other memory managers may be present that do not cause this automatic abort. If pmEasy reports this message, or if it does not get past the “STARTING IN REAL MODE” message, reboot with empty autoexec.bat and config.sys files and try again.

autoexec.bat and config.sys

Disk caches, memory managers, and other utilities can cause trouble for pmEasy. For example, one customer found that when he copied a new app.exe to his disk, the application would crash until he rebooted the system. The culprit was DOS’s SMARTDRV. Apparently, it was not flushing all of the newly copied file’s image to disk, so pmEasy was loading an inconsistent exe image.

If you have a problem with pmEasy or the application, reboot with empty autoexec.bat and config.sys files and try again. If that fixes it, put back the original files and remove lines one at a time until the problem goes away, to isolate the cause. See the Troubleshooting chapter for other possibilities.

Passing the Application Name on the Command Line

If PARSECMDLINE is set to 1 in PME.INC, pmEasy checks to see if an argument was passed on the command line. If so, it uses that as the name of the application to load.

Additionally, if DEFAULT_DRVXT is set to 1, pmEasy then checks to see if the drive letter and extension are specified in the name passed on the command line. If the drive letter is missing, pmEasy prepends the default drive letter (DefaultDrv) to the name. If the extension is missing, pmEasy appends the default extension (DefaultExt) to the name. These are set in UTILPME.C. This feature is intended as a convenience to save keystrokes. For example:

C:> **pme32 app**

would load app.exe from the default drive, which is A:, as shipped.

Binding pmEasy to the Application

pmEasy can be bound to the application. This means that the two programs are combined into a single .exe. The first part of the .exe is pmEasy and the second part is the application. This is a feature of the NE and PE file formats. The first part of the .exe is known as the real mode stub; the rest is the protected mode program. When the program is run from DOS, only the stub part is loaded into memory and executed. The protected mode part is invisible to DOS. Likewise, when a protected mode environment such as pmEasy loads the .exe, it sees only the protected mode part.[4]

Then, to run the program, it is only necessary to type the name of the application from the DOS prompt:

C:> **app**

This causes DOS to load and run pmEasy (the stub). pmEasy then re-opens the file and loads the protected mode part. All pmEasy needs to do is determine the name of the program it is bound to so that it can pass that name to its loader.

Of course, the name and path could be hard-coded into pmEasy, as is done with the appName variable. However, this is inconvenient since it will then only work if the application is on that drive and has that name. For example, if A:\APP.EXE is assumed, then it wouldn't work if you copied APP.EXE to the hard disk.

Luckily, DOS offers a way for a program to determine its own name. Better than that, it can determine the full drive, path, name, and extension. This is stored in a string at the end of the environment block, pointed to by a field of the PSP. pmEasy scans the environment block for the string and then passes it to the loader. The string is easily found because a marker precedes it. The string will vary depending on what drive and subdirectories the file is stored in. For example, it might be:

C:\DEMOS\PM\SMXDEMO.EXE

Typically, production systems will not have DOS (they will bootload directly to pmEasy), so this trick would be useful only during development. However, if the production system does have DOS, binding is an elegant way to ship your software since your application(s) appear to run as any DOS program would.

Building pmEasy for Binding

Pre-built copies of pmEasy for binding are supplied in PME\MISC\BIND. These are set with the default configuration options. If you wish to re-configure them, you need to rebuild following these steps:

1. PME.INC and PME.H:

Set BINDING_SUP	to	1
Set BIOS	to	1
Set BOOTLOAD	to	0
Set DOS	to	1

[4] Actually, for PE files, the stub is not totally invisible. For some reason, its size is included in the image size and all sections are shifted higher by this size. Our loader handles this, though, by shifting the sections back down so they immediately follow the section table, as they would if pmEasy were not bound to the application.

2. PME.MAK: Select DISKRD21 as the disk reader. Set DISKRD and any others to 0. DISKRD21 is necessary since it supports subdirectories; DISKRD does not. DISKRD would only successfully load the program if it were in the root directory. DISKRD21 uses DOS int 21h file i/o routines rather than implementing code to read the disk, so it is capable of loading from any media that is supported in DOS, unlike the pmEasy disk reader which supports a limited set of media. This also makes it smaller, so it adds less code to app.exe, resulting in a smaller file to transfer to the target or testing system.
3. Build pmEasy: "**mak r**" or "**mak d**"
4. Copy the resulting EXEs into \PME\MISC\BIND, replacing the default versions.

Setting BINDING_SUP = 1 enables the code that searches the environment block for the application name. It is possible to bind pmEasy even if this switch is not set, but this is not useful since then it will only be able to load whatever .exe is specified by appName, which may be a different program than the one you are trying to run.

How to Bind pmEasy to the Application

Binding is accomplished with the linker used to link the application. This section explains how to do this for several compilers. smx users should be pleased to know that the Protosystem already has built-in support for binding pmEasy. Enable the **bindpme** macro in the makefile. The makefile is set to bind a pre-built pmEasy from the PME\MISC\BIND directory. When building the Protosystem for debugging, the debug version of pmEasy is bound — the one that has the target monitor linked. When bindpme is disabled, a simple stub program is linked which displays the message "This program must be run under pmEasy32" (or "pmEasy16") if it is run from DOS. This stub program is kept in PME\MISC\BIND\STUB. It is useful for those not binding pmEasy, such as those bootloading from disk, since it identifies the app as a pmEasy app, not a Windows app, as the default stubs linked by the compilers' linkers indicate.

Borland and Microsoft 16-bit Compilers: The STUB statement is used in a linker definition file (.def) to tell the linker to bind the stub to the program. The .def file should contain, at a minimum:

```
PROTMODE
HEAPSIZE 4096
STUB '..\..\..\pme\misc\bind\pme16.exe'
```

The HEAPSIZE statement avoids a Borland warning that the heap is too small. It is not needed for Microsoft, but it causes no harm. All it does is set offset 10h in the Windows header of the .exe, which is ignored by pmEasy's loader.

Borland and Paradigm 32-bit Compilers: Again, the STUB statement is used, but the PROTMODE and HEAPSIZE statements are unnecessary and should be omitted. Note that single quotes are shown above because versions older than 5.0 give an error if double quotes are used, even though the documentation shows that syntax. Single quotes work for all compilers we have tried.

Microsoft 32-bit Compiler: This one is the easiest. A .def file is not needed; just pass the /STUB switch to the linker:

```
LINK /STUB:.\..\..\pme\misc\bind\pme32.exe
```

Other Notes about Binding

A program to which pmEasy is bound is equivalent to pmEasy by itself in the sense that it can be used to load a different program. For example, if pmEasy is bound to myprog1.exe, it could be used (if PARSECMDLINE == 1) to load another program, myprog2.exe, like this:

```
myprog1 c:\myprog2.exe
```

or if DEFAULT_DRVXT == 1, simply:

```
myprog1 myprog2
```

This follows from the precedence of options documented below. A name passed on the command line takes precedence over all other options. So in this case, pmEasy does not even consult the PSP to find that it is, in fact, bound to myprog1.exe. It loads myprog2.exe, as instructed. Whether or not this technique is useful, it is instructive.

Precedence of Options for Determining the Application Name

This section gives a concise summary of the procedure pmEasy follows to determine the name of the application to load. Note that the pmEasy loaders display the name that is ultimately passed to them, after conversion to upper case and any conversions done in the steps below. This makes it easy to see what the loader is attempting to load and should quickly reveal what is going wrong if the result is not as intended.

1. If PARSECMDLINE == 1 and an argument is passed on the command line the name passed is used, after the following conversions, if necessary:
   a. If DEFAULT_DRVXT == 1 and the name has no drive letter, DefaultDrv (UTILPME.C) is prepended to the name.
   b. If DEFAULT_DRVXT == 1 and the name has no extension, DefaultExt (UTILPME.C) is appended to the name.
2. If BINDING_SUP == 1
   a. The full program name is retrieved from the Environment Block, which is pointed to by the word at offset 0x2c of the PSP.
   b. If the program name does not begin with “pme” (case-insensitive) it is assumed that this executable is the application to load and that pmEasy is bound to it. This path and name is passed to the loader.
   c. If the program name does begin with “pme” it is assumed that this is pmEasy, so rule 3 is used.
3. If none of the conditions above are satisfied, the default name, appName, is used.

Transferring the Application to the Target PC

Removable Disk (Floppy, LS-120, IOMEGA Zip®, etc):

The application can be loaded directly from these disks or copied to the target’s hard disk. If using a floppy disk and the file is too large to fit, it can be compressed onto the floppy and then uncompressed onto the target’s hard disk. If using media other than those listed above, use DISKRD21. This disk reader uses int 21h services so it can read

from any disk for which you have a DOS device driver. The disk readers are discussed in the chapter *Loading the Application*.

Serial Link:

The DOS **INTERLNK** utility can be used to transfer files through the serial port to a target PC. The host must also be running DOS.

We are currently investigating the availability of utilities for transferring files from Windows machines to DOS machines over serial link and Ethernet (FTP). Please check the release notes for updated information on this topic. If you know of a good utility for this purpose, please email support@smxrtos.com.

Bootloading pmEasy from Disk with BIOS (no DOS)

pmEasy can be loaded via disk boot sector as a standard option. We provide a boot sector program and a simple utility that replaces the boot sector program on a disk with it so that it loads pmEasy. pmEasy then loads the application. DOS is completely out of the picture. The necessary files are in PME\MISC\BOOT. See the *Boot Sector* appendix for more information about the boot sector and boot sector program.

Utilities Used

1. **BOOTWRIT** replaces the boot sector program on a disk with one that loads pme.sys. It works for floppy disk, hard disk, and DiskOnChip®. It prompts for confirmation if the disk specified is other than a floppy, to help avoid accidentally altering your development system's hard disk. This utility and source code for the boot sector program are supplied in PME\MISC\BOOT. Syntax: **bootwrit a:**
2. **EXE2BIN** is used to convert pme16.exe or pme32.exe into a binary file. It prompts for the base address and makes fixups to all far addresses. This is a utility that was provided with DOS up through v5. It is also included in some versions of Windows such as 2000 and XP. At the command prompt, type "exe2bin /?" to see if it is present. The Tools page of our support site (www.smxrtos.com/support) shows sources for this utility or an equivalent. Syntax: **exe2bin <file.exe>**
3. **PAD** adds zeroes to the end of the binary file to force it to a certain size. This should be done to reserve space on your production system's disk for future upgrades of pmEasy, which may grow larger (it cannot be fragmented or it will fail to load). We supply this utility in PME\MISC\BOOT. Syntax: **pad <file> <size in KB>**

Making a Bootable Disk

These are the steps to create a bootable disk or diskette so pmEasy runs from system boot. If you have problems, please consult the readme file in PME\MISC\BOOT for any additional notes or changes to the procedure below:

1. PME.INC and PME.H:

Set BINDING_SUP	to	0
Set BIOS	to	1
Set BOOTLOAD	to	1

Set DEFAULT_DRVXT	to	0
Set DOS	to	0
Set PARSECMDLINE	to	0

2. PME.MAK: Select DISKRD (not DISKRD21) as the disk reader. Comment out all drivers except the one needed since EXE2BIN will not convert the .exe if it is too large.
3. Build PME16.EXE or PME32.EXE: "**mak r**"
4. Convert to binary. For example: "**exe2bin pme32.exe**"
   Supply "**800**" (hex) at the fixups prompt. EXE2BIN is a utility provided with DOS but alternatives may be available. (See the *Utilities Used* section above.) The entry point, reported at the end of the .map file, must be 0:0 or EXE2BIN will report "Cannot convert file."
5. Rename PME16.BIN or PME32.BIN to PME.SYS.
6. Pad PME.SYS to 64KB: "**pad pme.sys 64**". This reserves space for updates. See the notes for PAD.EXE below.
7. Format the disk or diskette you want to bootload pmEasy from, using DOS or Windows.
8. Replace the boot sector program: "**bootwrit a:**" or "**bootwrit c:**" . If writing to the hard disk on the target, you need to copy BOOTWRIT.EXE and BSEC32.BIN or BSEC.BIN to a floppy **to run on the target**.

   If you have just accidentally overwritten the boot sector on your development system's hard disk, this may be difficult to reverse, but see *Restoring the Boot Sector* in the *Boot Sector* appendix for tips.
9. Copy PME.SYS to the disk or diskette. It must be one of the first 16 directory entries. That is, it must be in the first sector of the root directory. This permits the disk to have a volume label (which occupies the first directory entry). Windows NT users, please see "WINNT Note" below.
10. Copy APP.EXE to the disk or diskette.
11. Reboot the computer.

More about the Utilities

BOOTWRIT.EXE replaces the boot sector program on a DOS/Windows-formatted disk with one we developed. Source code is supplied in PME\MISC\BOOT\BOOTSECT. Note that 2 boot sector programs are required: one for FAT12/FAT16 disks and one for FAT32 disks. This is necessary due to differences in the formats of these disks. Both boot sector binary files are generated from the same source file, **BOOTSECT.ASM**.

BOOTSECT.ASM and BOOTWRIT.C are provided so you can modify their behavior as desired. BOOTSECT.ASM is a tricky program to work on because space is so tight. The entire boot program must fit in about 450 bytes (420 for FAT32). Some routines are provided for basic output of markers to the screen but there is only enough space to enable a few markers at a time. This boot program is more capable than the DOS boot sector since it allows the program it loads (pme.sys) to be anywhere in the first 16 directory entries.

CAUTION: Be careful not to run "bootwrit c:" on your development system or it will no longer boot. If you have already made this mistake, see *Restoring the Boot Sector* in the

Boot Sector appendix. Micro Digital assumes no responsibility for damage you do to your hard disk or other disk, if you use this utility.

PAD.EXE makes a file longer by appending zeros. The new size is specified on the command line. You should pad pme.sys to 64KB to allow for updates in the future. This is necessary because pme.sys must not be fragmented. If you need to install an upgrade to units in the field, the new pme.sys must not be larger than the original or it will not fit in the space occupied by the old pme.sys.

Do not pad to more than 64KB, since this is the largest file the boot sector can load — any sectors beyond this "wrap around," overwriting the code at the beginning. This is because the destination pointer ES:BX starts at 800:0 and ES is never advanced, making 64KB the maximum range.

The advantage of this padding utility as opposed to adding pad bytes to the code is that it can be run from a batch file and it will always specify the same size — there is nothing to manually adjust as pmEasy changes size. This utility can be easily changed to accept the size in bytes rather than KB — see comments in PAD.C.

WINNT Note: If your development machine is running NT and you want to update PME.SYS on a boot floppy, you will find that eventually, it will no longer bootload. This is because NT preserves deleted directory entries and uses a new one each time a new file is copied. Eventually all 16 root directory entries will be used, so PME.SYS's directory entry will be in the second sector of the root directory. It will no longer be found by the boot program, since it scans only the first sector of the root. To solve this problem, reformat the diskette.

Running pmEasy from ROM

pmEasy can be located and put in ROM, to load the application as an .exe from disk. Although we do not provide this as a standard option, it could be achieved without too much effort using a locator such as Paradigm LOCATE or CSi-Locate and the locator startup code to handle copying pmEasy’s initialized data to RAM and clearing uninitialized data segments (BSS and c_common). It would be necessary to write whatever hardware startup code is required unless the target has a BIOS that handles this. In this case, pmEasy could be made a BIOS extension. Note that to run the pmEasy code from ROM, a full-fledged locator must be used rather than the EXE2BIN utility, because data and code must be located to separate address spaces and EXE2BIN is capable only of generating a sequential binary image for execution in RAM. Another option is to locate the entire pmEasy image for RAM with EXE2BIN and write a little boot program to put at the reset vector that copies the image from ROM to RAM and jumps to it.

Note that the DPMI 300h routine must be located below 1MB since it switches to real mode. Most users do not need this routine, but for those that do, if there is ROM only above 1MB, the easiest solution would be to use EXE2BIN as discussed above, rather than trying to locate this code separately from the rest of pmEasy.

Restarting pmEasy

pmEasy can be restarted to run from the beginning. This is a useful feature for debugging, since it allows restarting the application from the debugger without having to reboot the

target and re-run pmEasy. This option is controlled by the RESTART setting in PME.INC. By default, it is enabled for the debug version of pmEasy.

The restart or reload command of the debugger sends a command to the debug monitor. The handler for that command has been modified to jump to *pmRestart*. The code at *pmRestart* switches back to real mode and jumps to *pmInit*, which is pmEasy's entry point. pmEasy re-initializes itself and reloads the application from disk.

If a hard disk can be connected to the target during development, this offers a very fast debug cycle and good reliability, since all data and code are reloaded. Using a traditional embedded approach, a reload means downloading the application over the serial link, which can take a significant amount of time. A reload from the target's hard disk is almost instantaneous. Because serial downloads are slow, debuggers usually offer the option of reloading just the registers and resetting the CS:EIP to the program's entry point, without actually reloading the program. This can mean an unreliable restart if data or code in the program were corrupted. Restarting pmEasy and reloading the application from disk, by contrast, ensures a reliable restart.

In order to make this work, the _DATA segment is copied into an area in RAM the first time pmEasy runs so it can be restored to its initial values on restart. See the *Buffer Area* section of the pmEasy *Internals* chapter for more information.

Preparing for Release

Although the messages pmEasy displays are convenient during development, they may not be desirable in your production system. All video output and keyboard waits can be disabled by configuration options. Please refer to the keyboard, loader, and video settings in the chapter *Configuration Reference* for discussion of all options.

Loading the Application

Loading the Application from Disk

pmEasy has the ability to load your protected mode application from disk and start it running. Your application must be linked in the New Executable (NE) file format for pmEasy16, or in the Portable Executable (PE) file format, for pmEasy32. NE is the standard format used for 16-bit Windows applications and PE for 32-bit Windows 9x and NT applications. Both of these file formats are readily produced by the linkers provided with the Borland, Microsoft, and Paradigm C/C++ compilers. If you are interested, a discussion of each format is presented in the *File Formats* appendix of this manual. Directions for preparing the executable and notes about these file formats are in the sections that follow. It is also possible for the application to be in hex format, for pmEasy32, but this is not a standard option. See the *Hex Loader* section below.

Initially, pmEasy is configured to load the application from floppy disk (A:). To load from hard disk, change *appName* in PME.ASM. See the *Drive Lettering* section below. The previous chapter, *Running* pmEasy, discusses options for passing the program path and name on the command line and binding pmEasy to the application.

Media Supported

pmEasy currently supports loading the application from the following media:

- Floppy
- IDE Hard Disk (FAT16 and FAT32)
- LS-120
- IOMEGA Zip® Disk
- ATAPI CD-ROM
- M-Systems DiskOnChip® 2000 and Millenium Flash Disk
- DOS Disks (when using int 21h disk reader)

Both 3.5” and 5.25” floppy disks are supported. For IDE hard disks, pmEasy automatically detects whether they are FAT16 or FAT32; there is no configuration option to set. Note that for ATAPI devices such as CD-ROM, LS-120, and Zip® drives, there is a 10- to 15-second delay before the load starts, due to the initialization sequence required by these drives. Use of DiskOnChip is covered in the *DiskOnChip* appendix. It indicates which DiskOnChip devices are supported.

Please refer to the release notes (e.g. PME22.TXT) to see if other drivers have been added since this manual was last updated. Other media can be used if DOS is present — see discussion of DISKRD21 in the *Disk Readers* section below.

Disk Readers

The disk readers are stored in the PME\LOAD directory.

DISKRD: This is a simple disk reader. It was developed by Micro Digital to be as lean as possible. It has only enough capability to handle the loader's requests to read the .exe file from disk. DISKRD uses the drivers from smxFile/ERTFS, with some modifications. These communicate directly with the hardware. Subdirectories are not supported. Source code is provided for the disk reader but not the drivers, unless purchased separately or with smxFile. This is the disk reader to use when bootloading pmEasy without DOS. To enable this disk reader, set DISKRD to 1 in PME.MAK and all others to 0.

DISKRD21: This is a variant of DISKRD that uses DOS int 21h services to load the application. DPMI 300h is used to switch to real mode to issue the int 21h calls. This approach is useful for loading from media unsupported by DISKRD, during development or in the final target, if DOS is present. Any device that can be read from DOS is supported. Subdirectories are supported. Full source code is provided. Since all the work is done by DOS there is not much code; this is the smallest disk reader. This is the disk reader used when binding pmEasy to the application. To enable this disk reader, set DISKRD21 to 1 in PME.MAK and all others to 0.

RTFS: This is the disk reader used in pmEasy v1. It supports only floppy and IDE loads. It is not supplied with pmEasy v2, but currently the makefile still allows enabling it, by setting RTFS to 1 and all others to 0.

Only DISKRD21 supports subdirectories. If an attempt is made to load the application from a subdirectory using the other disk readers, the loader will attempt to find the file in the root. If not there, the load will abort with the message Error opening file.

These disk readers do not support disk writes and are not intended for application use. You can purchase a full file system from Micro Digital for use in your application.

Drive Lettering

Drive lettering depends on which disk reader is used:

DISKRD21 drive lettering is the same as DOS. Whatever letter DOS knows the drive by is the letter that should be passed to pmEasy.

DISKRD drive lettering follows this simple scheme:

A:, B:	Floppy
C:, D:, E:, F:	IDE hard disk, LS-120, Iomega Zip® or ATAPI CD-ROM (E: and F: are assumed to be CD-ROM, by default. See bold notes below, to change this.)
G:, H:	DiskOnChip® 2000 or Millenium (M-Systems flash disk)

This is true even when using only one driver. Stubs are linked in place of drivers that are commented out in the makefile. These stubs return FALSE when called. For example, on attempt to load from A:, if the floppy driver is not linked, the stub will return FALSE when the loader calls it, and the load will abort, as expected.

Each drive letter corresponds to a different device; partitions are not supported. Only the first partition on a disk is seen. For example, C: corresponds to the master IDE drive on the primary controller. D: is the slave on the primary controller.

We suggest you continue to use this lettering scheme. However, it can be changed by modifying the settings in DISKRD\pcconf.h and rebuilding DISKRD clean.

CD-ROM is a special case since it is read by a separate file system which is called by DISKRD if the drive number is in the range set for CD-ROM drives. **As shipped, the disk reader assumes there is 1 CD-ROM drive and that it is E:**, meaning it is connected as the master on the secondary IDE controller. If you wish to connect it differently, such as to the primary controller, set FIRST_ATAPI_CDROM_UNIT in LOAD\DISKRD\DISK.C and rebuild the disk reader. Assuming there is only 1 CD-ROM drive saves a significant amount of space for data structures. If the target has more than 1 CD-ROM drive, and you wish to be able to load from either with the same pmEasy executable, you must set N_ATAPI_CDROM_UNITS and rebuild the disk reader and CDFS. Source for CDFS must be purchased, to rebuild. (The alternative would be to set FIRST_ATAPI_CDROM_UNIT differently in 2 builds of pmEasy and use the appropriate version depending on the drive to load from.)

Those not loading from CD-ROM should set ENABLE_CDROM_CODE to 0 in disk.c, so that E: is no longer assumed to be a CD-ROM drive. Doing this allows loading from a non-CD-ROM drive connected as master on the secondary controller. This also saves a little code.

Driver Selection for DISKRD

To keep pmEasy small, link only the drivers you need. Drivers are selected in the pmEasy makefile, in the section for the simple disk reader, near the top. Comment out the lines for drivers you don't want.

```
# Select Drivers for DISKRD
#floppydriver = ..\..\load\diskrd\fl_drver.obj
#idedriver = ..\..\load\diskrd\ide_drv.obj
#msysdriver = ..\..\load\diskrd\msys\msys.lib
...
```

Stubs are linked in place of missing drivers, which return FALSE on attempt to load from those media.

Because the debug version of pmEasy links the debug monitor, it has space for only a couple drivers.

Loading from Other Sources with DISKRD

We have achieved a nice separation between the loaders and the disk reader, so that you can make pmEasy load from alternate sources without needing to modify the loader. Instead, alter DISKRD or create a new disk reader based on it. DISKRD\disk.c shows clearly the 7 API routines that are needed by the loaders. These can be re-implemented to get the data from any source, such as linear ROM/Flash or data link.

The routines of primary importance are po_read() and po_lseek(), which read data from the file and seek to a new offset in the file, respectively. As long as these pass back the data the loader requests, the loader does not care where it came from.

Loading app.exe from linear ROM/Flash should be relatively simple — just copy the bytes from the current file position into the destination buffer. To load from a data link, it would be necessary to implement the communications protocol.

Tip: Whenever adding code to pmEasy, be careful about far pointers. See *Altering the Code* in the *Customizing* pmEasy chapter of this manual for more discussion.

Exe Loaders

pmEasy16 and pmEasy32 have loaders for the New Executable and Portable Executable file formats, respectively. Full source code for the loaders is now provided. There is no reason to alter the loaders, except in unusual circumstances. Supporting alternate media can be achieved by altering DISKRD, not the loader. See the section *Loading from Other Sources with DISKRD*. See the *Configuration Reference* for settings that can be changed. The source code is useful if it is necessary to fix a problem, such as an unexpected record type, or as a guide to write a new loader for another executable file format.

The loaders allocate memory for the application from pmEasy's heap. For NE loads, descriptors are allocated from the LDT for all segments in the .exe. The .map file shows the segments and how many there are. For PE loads, only 2 descriptors are needed: a flat code and flat data segment. These span the entire 4 GB memory space. They are allocated from the LDT, in PMI.ASM at startup, not in the loader.

Both loaders use the entry point stored in the executable's header as the first instruction to execute in the program. (In versions of pmEasy before v2.1, the PE loader ignored the entry point field and assumed the entry point was at offset 0 of the code section of the program.) See the section *Startup Code and Entry Point* in the *Application Development* chapter for further explanation.

The loaders assume that the stack pointer will be set by application startup code. Fields in the executable's header related to the stack pointer or stack allocation are not used.

An enhancement made to the loaders for pmEasy v2 was to allocate all temporary memory used to hold the headers and other structures at the top of pmEasy's heap. This reduces heap fragmentation and is especially important for those who are using smxDLM, since many DLMs might be loaded — with this feature, the DLMs will stack one after the other instead of having gaps between them. Additional enhancements have been made to the PE loader. See the section *PE Loader Features*, below for discussion.

Note for pmEasy v1 users: Because of the enhancements made to the NE and PE loaders, the base selector (NE) or base address (PE) of the loaded application will be different than in pmEasy v1. In order for debug symbolics to match the code, the new APP16.CMD or APP32.CMD (in PME\CSIMON\CMD) must be used with CSi-Locate since it specifies the new base selector/address. pmEasy32 users who change HEAP_START must adjust the address in APP32.CMD accordingly.

If CHECKSTACK is 1 in PME.INC, pmEasy does a stack check after the load completes to ensure there has been no stack overflow or underflow. Then it prints the percentage of stack usage, in decimal (all other pmEasy output is in hex). The loader and disk reader are the primary stack users so it makes most sense to check the stack after the load. The drivers use varying amounts of stack, so depending on which is used, it may be possible to reduce PMSTACK_SZ in PME.INC. For example, with a 1300-byte stack, we observed these stack usages after loading from the following devices:

Floppy:	33%
IDE Hard Disk:	62%
CD-ROM:	88%
DiskOnChip:	42%

NE Loader Special Features

The NE loader has remained mostly unchanged for pmEasy v2. One new feature is that temporary blocks of memory are allocated from the top of the heap (blocks are normally allocated searching from the bottom). This reduces heap fragmentation. Similarly, the temporary LDT descriptors used to map these blocks are allocated from the top of the LDT.

PE Loader Special Features

Several enhancements have been made to the PE loader for pmEasy v2. As in the NE loader, temporary memory is allocated from the top of the heap and temporary descriptors from the top of the LDT. Other nice improvements have been made, as discussed below.

The **.debug** section of a PE file contains debug symbolics used by the debugger. However, there is no need to load this information into the target's memory, since it has already been extracted on the host for use in the debugger. pmEasy does not load this section, which greatly reduces load time. Sometimes there are more bytes of debug symbolics than code and data in the program! Note that starting with Visual C++ 6.0, Microsoft's linker no longer puts the debug symbolics into a section, so they are invisible to the loader anyway, and this feature of the PE loader is not needed. However, it is still relevant for Borland and Paradigm users and those using older versions of the Microsoft compiler.

The **.reloc** section contains fixup information. Once the fixups have been performed the section is no longer needed, so the loader discards it, freeing the memory it occupied, making more available to the application. Like other temporary allocations, it is allocated from the top of the heap to reduce fragmentation.

As in pmEasy v1, if it is unnecessary to perform fixups, due to the preferred base being set to the actual address where pmEasy loads the application, pmEasy does not load the .reloc section and skips the fixup phase, speeding the load. In v1, though, the loader would abort if no fixup table were present in the program. Now, provided the condition is met, the fixup table need not be present, which means you can reduce the .exe size as well, if that is an issue. (If you want to set the preferred base to take advantage of these optimizations, you should not bind pmEasy to the application. See the section *Notes about Binding* pmEasy *to a PE* in the *File Formats* appendix of this manual for explanation.)

The loader allocates a little extra space beyond what is needed for the program image so that the loader can start the image on a 256-byte boundary. This is required by the GoFast™ coprocessor emulator, since its emulation stack must be aligned on such a boundary. We considered using the traditional 4KB section alignment, but this is unnecessarily wasteful, especially for smxDLM users who load many DLMs. We feel 256 bytes is the largest boundary that should be needed, since this is the largest alignment that can be specified in assembly language. If a larger alignment is needed (we would be curious to find out why), it can be easily changed by setting the ALIGNMENT constant in the loader source code. If you do not have loader source code, please contact Micro Digital.

Loader Screen Display

As shipped, the loaders display some interesting bits of status information to the screen. This is discussed in the subsections which follow. These messages are helpful during development, but we realize that you may not want them in the final system, so we provide 3 loader .obj's to choose from, which display varying degrees of messages. Select the one you wish to link, in PME.MAK.

Note that the pmEasy VIDEO conditional is used to quiet all screen writes. Even though setting this to 0 will quiet the loaders, you will want to link a quiet version of the loader since they are smaller and faster (since the calls to the screen write routines were conditioned-out when they were compiled).

(Before we began to provide 3 pre-configured loader .obj's, we quieted the loader by setting the *quietLoad* variable. Everywhere in the loaders, this variable is consulted before writing to the screen. We have retained this approach in case a user has the need to quiet the loader dynamically — unlikely, maybe, but it didn't cost us anything to keep it.)

NE Loader Display

The NE loader displays information about the **segments** loaded and indicates for each its number (which corresponds to the .map), selector, whether it is code or data, its size, and its base address. The **image size** is the total size of all segments and is calculated by adding the size of each segment as it is allocated. The **entry point** is the address of the first instruction of the program.

PE Loader Display

The PE loader shows information about the **sections** loaded and indicates for each its number, name, base address, virtual size, whether it is code or data, and any additional notes. The **base address** is the linear address where the section starts in memory. The **virtual size** is a field in the entry for each section in the section table. This is the value you will see if you run DUMPBIN or TDUMP on the .exe and look at the section information. This field has a slightly different meaning depending upon which linker is used. Microsoft uses it to mean the exact number of bytes used in the section; padding to the next section alignment boundary (i.e. 4KB) is ignored. Borland includes the padding in the value it specifies.

Currently, the entries you are likely to see in the **notes** field are "used, discarded" and "not loaded". The former is displayed after the loader has finished performing fixups and the .reloc section is no longer needed. The latter is displayed for the .debug section, always, and for the .reloc section when the preferred base matches the actual base address where pmEasy loads it. See the *PE Loader Features* section, above, for more information.

The **image base** and **preferred base** are displayed. The image base is the actual address that pmEasy loaded the program to. The preferred base is what was specified to the linker. The linker sets all addresses relative to this value, effectively locating the program for this address. The **bound program size** is displayed as a reminder not to bind pmEasy to the application if the goal is to avoid performing fixups by setting preferred base == image base. When these are equal and when bound program size is 0, the image base and preferred base values are shown in bright white, indicating that it is unnecessary for pmEasy to perform fixups, so this step is being skipped.

The **total size** is the size reported in the image size field of the PE optional header. This includes the stub program, .reloc, and .debug. The **image size** value shows the size with these removed. The **alloc size** is how big a block was actually allocated to hold the program. This includes a little extra space so that the loader can start the image on a 256-byte boundary. This is required by the GoFast™ coprocessor emulator, since its emulation stack must be aligned on such a boundary. See the *PE Loader Features* section, above, for more discussion of the image alignment.

Finally, the **entry point** is shown, indicating the address of the first instruction of the application. This is the entry point field in the PE optional header, adjusted by the image base address. (In pmEasy versions before v2.1, this was just offset 0 of the application image.)

Producing the NE and PE Formats

NE Format (pmEasy16)

Producing a New Executable (NE) is accomplished by building with the 16-bit Borland or Microsoft C/C++ compiler. Use the /G2 or /G3 option (Microsoft) or –2 or –3 (Borland and Paradigm) and supply the linker with a .def file containing the following line:

```
PROTMODE
```

This is all that is necessary. See PM.DEF and the makefiles for the sample applications provided with pmEasy. Note: smx users who wish to use the /G3 or –3 switch must purchase smx source code in order to rebuild the smx library for this option.

*PE Format (***pmEasy32***)*

This is the default .exe type produced by the Borland, Microsoft, and Paradigm 32-bit linkers supplied with their C/C++ compilers. No special steps are required. See the makefiles for the sample applications provided, in the BC.P3 or MC.P3 subdirectory.

Hex Loader

Another option for loading the application is a Hex loader. It is currently supported only for pmEasy32.

The Hex file format is an absolutely located format, which is produced by a linker/locator. For this reason, it is not as convenient as the EXE format and not especially suited to the spirit of pmEasy. Also, because of the representation of each binary byte as 2 ASCII bytes, the file is roughly twice as large as the corresponding EXE would be. Nevertheless, this is a useful format since it is checksummed. Typically, each 32 bytes of the file is followed by a checksum. This makes it good in systems where a new program image is downloaded to the target's disk over a data link, for example, and there is concern about the reliability of the transfer.

The Hex loader displays the number of the record it is currently processing. Also, error messages indicate the record number that the error pertains to. Note that numbering starts at 1 to correlate with the line numbers in an editor. Linker/locators typically start each record on a new line and editors typically number the first line as 1. So if an error occurs, the record number reported is the line number to go to in the editor to see where the failure

occurred. As with the other loaders, 3 pre-built versions are supplied with varying amounts of screen output from all to none.

If using the Hex loader, there are several constants in PME.INC that must be set to tell pmEasy information about the application being loaded, such as its entry point and where the LDT is stored in it. See the constants adjacent to APP_ENTRY_OFF in the HEXLOAD conditional. A buffer is used to store the ASCII data being loaded. Its size is controlled by HEXLD_BUFSIZE in PME.INC. By default it is set to its maximum size (almost 64KB) since it is allocated from the heap and freed when the load completes. There is probably no reason to change this setting. Set the extension for appName to .HEX in PME.ASM.

Application in ROM

If your goal is to produce a located application to run from ROM/flash, you should use a linker/locator rather than pmEasy. We recommend Paradigm C++, which is a full tool suite, including compiler, assembler, linker/locator, debugger, and IDE.

If you prefer the simplicity of EXEs and pmEasy, but you would like to store the image in ROM either to protect it or because the target system has no disk drives, you can do this with pmEasy. A good choice is to use an M-Systems DiskOnChip since pmEasy has built-in ability to load from it, and it is easy to copy the program to it since it is accessible as a normal disk in DOS. See the *DiskOnChip* appendix.

If, instead you have some other ROM or flash part in your target, you can program the EXE image into it and use the pmEasy loader to load it into RAM, as usual. To do so requires modifying the disk reader, DISKRD, to copy the bytes from ROM/flash into RAM. See the section *Loading from Other Sources with DISKRD*, above.

Application Development

Where Do I Start?

If you are using smx, start from the Protosystem. See *Part II: Application Development* in the smx Quick Start. Ignore discussion of MINCAPP and MINAAPP; skip ahead to the section *Your Application*.

Otherwise, start with MINCAPP and MINAAPP. These are simple C and assembly applications that you can immediately build with the makefiles supplied and run under pmEasy. Do that. This ensures that you are using the same development tools we are and that your development system is set up properly. This is sometimes not a trivial accomplishment, and you should breathe a sigh of relief once you have the pure versions building and running fine. If you have build problems, please see *Tools for Building the Application* in the *Tools* chapter, earlier, to confirm that you are using the same versions of the tools we recommend.

Building the Sample Applications

Makefiles are provided for the sample applications. Both are built the same way. From the directory PME\MINCAPP\MC.P3 (or similar) type:

pmEasy16: **mak l r**
pmEasy32: **mak r**

The “l” means large memory model. The second argument is the version, which can be release (“r”) or debug (“d”) .

Note that different memory models are possible only for pmEasy16 applications, since pmEasy32 applications are flat memory model, only. For pmEasy32, the memory model argument is omitted. Although different memory models are possible for pmEasy16, we have tested only large model, since that is most suitable for large applications (which is what we expect most 16-bit protected mode applications will be since that is the main reason most users choose 16-bit protected mode over real mode).

A directory is created, under which the executable and object files are placed. If you typed “mak l r”, the directory will be LR (\PME\MINCAPP\MC.P2\LR, for example). For 32-bit flat mode, you do not specify the memory model, but we name the directory SR, for example. The “S” designates small/flat model.

In this way, object modules for different versions are put into different directories. Regardless of whether you continue to use LR and LD as the names or RELEASE and DEBUG, we recommend this approach since it is more convenient and less error-prone than building .obj modules for all versions in the same directory. The problem with having only 1 directory for .obj modules is that you must remember to clean out the old .obj’s

before switching between a debug and release build. If you forget to delete all the .obj's only some may build causing unexpected results (since some modules will be built for debug and others built for release). Build time is also reduced by leaving it to the make utility to decide what to rebuild rather than doing a clean rebuild. Also, keeping the .obj's separate from the sources is cleaner and easier on the eyes when listing the files in the directory.

If you wish to re-organize the directory structure, please keep in mind that all of our makefiles use relative paths, so any significant changes to the directory structure will necessitate changes to the makefiles. We recommend that you start with our makefiles, but if you prefer to write your own, **be sure to use all of the same compiler and linker switches!** Leaving out a necessary switch is sure to cause trouble.

Running the Sample Applications

Now that you have built the sample applications, you are ready to run them. Copy PME32.EXE and APP.EXE to a floppy or hard disk on the target PC. Then type one of these commands (pmEasy16 users, substitute "pme16").

```
C:>pme32                (loads A:\APP.EXE)
C:>pme32 c:\app.exe
```

Please refer to the first section of the chapter *Loading the Application* for more information about running pmEasy and the application.

Here is what you should expect to see when running the sample applications.

MINAAPP (Minimal Assembly Application)

This example only displays an "A" in the upper-right corner of the screen, and exits to pmEasy. Other than the screen output, the notes for MINCAPP, below, apply here too.

MINCAPP (Minimal C Application)

This example just displays the messages "APPLICATION IS RUNNING" and "APPLICATION DONE". It immediately finishes, so you see pmEasy's APP DONE message at the top. The return code of 0 means that execution was successful. start__.asm loads an error/completion code in AX just before returning to pmEasy. You can use this mechanism to return an error code should your application fail.

You can see from the source code what is minimally required for an application — not much! If you wish to try writing your own code, we suggest you start with this example. Use the makefile provided, and modify it as necessary.

Your Application

Your application should be a Windows console application, not a GUI application. Notice that WinMain is stubbed off in the sample apps' startup code.

Remember that Windows and DOS aren't present, and this can cause problems when calling C run-time library functions that are OS-dependent, such as screen or file i/o. Microsoft, Borland, and Paradigm 32-bit C libraries are implemented to use Win32 services. The 16-bit libraries use DOS services, mainly int 21h. When an int 21h call

occurs, pmEasy displays the message "MSDOS CALLS NOT SUPPORTED…". Win32 references are caught by the linker. Borland users should remove import32.lib from the link line in the makefile to see if any Win32 references are being made. This library satisfies the linker but leaves dangling pointers to non-existent functions (since there are no Windows DLLs present). This is ok if the functions are called in paths that never execute. You should stub off or re-write any such functions that are needed, or copy sources of needed C library functions into your application and modify them to remove the Win32 dependency. Our *unWin* product replaces many Win32 functions. Although developed for use with smx, it could be adapted for use in a non-smx application too.

pmEasy32 users with an old monochrome display should define MONO in CRT.C to ensure the right video address (B0000) is used.

Module Inclusion Order

If using smx, use the same inclusion order for header files as CORE.C uses, in the Protosystem.

If starting with MINAAPP:

include app.inc	Contains various environment-related definitions.
include pme.inc	Main pmEasy include file.

If starting with MINCAPP:

#include "app.h"	Contains various environment-related definitions.
#include "gen.h"	Contains typedefs and other basic definitions.
#include "pme.h"	Main pmEasy include file.
#include "cpmi.h"	C API for DPMI functions.
#include "crt.h"	Video routines

Defines Used in Makefiles for MINAPPs

Except for the PME16 or PME32 define, all defines (/D or -D) in the makefiles are suggestions. These makefiles originated from the smx Protosystem makefiles, and we kept some of the defines with the thought that you might find them helpful. Some are needed, as shipped, but the notes below explain the minor steps required if you wish to remove them.

Assembly and C Modules:

DEBUGVER: Used to control differences in code for a debug/development build vs. a release build.

PME16: Must be defined for 16-bit pmEasy applications.

PME32: Must be defined for 32-bit pmEasy applications.

Assembly Modules:

BORLANDC, **MICROSOFTC**, **MASM**, **TASM**: These are intended to handle differences between compilers and assemblers. You may delete the sections of code for tools you are not using and remove these defines from your makefile.

_MM_: Memory Model. Used to set large_code and/or large_data in APP.INC. This is useful to make code conditional to match the memory model used for C modules. If you are sure of the memory model you wish to use, you may delete the conditional code for other memory models and remove this define from the makefile.

Startup Code and Entry Point

pmEasy applications need only a minimal startup code to initialize the segment registers, to set the stack pointer, and, optionally, to clear the uninitialized data segments (_BSS and c_common). You may need to add some application-specific initialization code of your own. We purposely do not link the C library startup code because it is operating system dependent. Specifically, it makes DOS or Windows calls. You should, however, study it and use it as a guide. Copy definitions or routines needed by C library functions you want to use, when the linker complains of their absence.

pmEasy16

The 16-bit sample applications provided with pmEasy all keep out the C library startup code and instead use their own. See the start.asm modules of the sample applications.

Keeping out the Borland startup code is done by not linking the startup code object module. Unfortunately, for Microsoft tools, there is no such simple mechanism. The symbol __acrtused brings in the startup code. At a minimum, you must define __acrtused to resolve the references to it. This is usually not sufficient, though, since many of the routines in the Microsoft C library reference other variables and routines in the startup code, which will bring it in, even if you have defined your own __acrtused.

As you add code, and particularly, as you add C library calls, it is likely that you will introduce new references to the startup code, which will cause it to be linked. It will be clear that this is happening, since you will get duplicate symbol errors for the dummy startup code symbols you have already added. The strategy in locating the cause of the new problem is to first save a copy of the C library you are linking and then remove crt0 (the startup code), rebuild and find out what is bringing it in. You may find you need to remove more modules from the library to get at the ultimate source of the problem. Eventually, you should be able to determine which variables or routines in the startup code are required. Look at the startup code to see if you can put the same definition or a dummy routine in your own startup code.

Keep in mind that you may not be able to use some C library routines without the startup code — unless you take the trouble to learn what the routine requires and implement it in your own startup code.

For pmEasy16 applications, declare your entry point after the END statement in your startup assembly module, as usual.

pmEasy32

As with pmEasy16, you should not link the compiler startup code, but use it as a guide and a source of definitions needed by library functions.

Starting with v2.1, pmEasy32 no longer assumes the application entry point is at 0. Instead, it gets the starting address from the EntryPoint field of the PE Optional Header. Prebuilt

executables built for the old pmEasy probably will not run in v2.1 and later, since they are unlikely to specify the proper entry point. If that is the case, pmEasy will attempt to run the program starting at whatever address the entry point field specifies. Starting with v3.5 of smx, the Protosystem is set to specify the entry point. To use an older version of smx, it is necessary to specify the entry point after the END directive in START32.ASM, like this:

```
END     _c_startup        ; specifies app entry point
```

When we originally developed pmEasy32, we had implemented it to use this field to get the application entry point, but had problems getting a link with early versions of Borland C/C++ to set the correct entry point in the header. Now that we are able to get it to work, the main motivation for making this change was for the benefit of Developer Studio and the Borland IDE, to avoid the need to control link order. No longer must we ensure that startf.asm in the smx Protosystem is linked first.

DOS Emulation

The 16-bit C run-time library and 16-bit third-party libraries often are written assuming presence of DOS or a DOS extender. In order to minimize changes to such libraries, it is possible to emulate the DOS services they require, in protected mode routines. pmEasy provides DOSEM.C for this purpose. This is a stub module in which you can implement DOS int 21h routines needed by your application. By default, pmEasy hooks the *msdos* ISR (see PME.ASM) to int 21h. This, in turn, calls dos21Em() in DOSEM.C. Our unDOS product implements many int 21h services. Although developed for use with smx, it could be adapted for use in a non-smx application.

Win32 Emulation

The 32-bit C run-time library uses Win32 services for OS-dependent functions such as file i/o and screen i/o. (Simple routines such as memcpy() do not.) Use of a function that uses Win32 services will usually result in an unresolved symbol link error. To solve this, implement or stub off the Win32 service or modify the C library function to remove the Win32 dependency (if source code is provided). Our *unWin* product implements and stubs off many Win32 services. Although developed for use with smx, it could be adapted for use in a non-smx application.

Writing ISRs

A key point is that the 386 supports both 16-bit and 32-bit interrupt and trap gates. The *type* field of the interrupt or trap gate descriptor specifies which type of gate it is (see Table 6-1 in the *386DX Programmer's Reference Manual*). You control this for ISRs you create. The difference is that 16-bit gates cause 16-bit Flags, CS, and IP to be pushed, while the 32-bit gates cause 32-bit values for these to be pushed (the high word for CS is undefined). For this reason, 16-bit ISRs must do an IRET, while 32-bit ISRs must do an IRETD. Note that the difference between interrupt and trap gates is that interrupt gates disable interrupts, while trap gates do not.

16-bit C compilers, such as Borland and Microsoft, create 16-bit ISRs. That is, the prolog generated for a function defined with the *interrupt* keyword saves all the 16-bit registers and the epilog contains an IRET. 32-bit ISRs must push 32-bit registers and end with

IRETD. Unfortunately, the Microsoft and Borland 32-bit compilers do not support the *interrupt* keyword, so writing an isr for them takes a bit more effort.

Microsoft (32-bit) introduced the *naked* attribute for writing ISRs. It causes the function to have no prolog or epilog. You must write these yourself using inline assembly at the beginning and end of the routine, as shown:

```
#define DATA_SEL 0x14  /* data selector assigned by pmEasy loader */

#define ENTER_ISR() \
        { \
        _asm  pushad \
        _asm  push  ds \
        _asm  push  es \
        _asm  cld \
        _asm  mov   ebp,esp \
        _asm  sub   esp,__LOCAL_SIZE \
        _asm  mov   ax,DATA_SEL \
        _asm  mov   ds,ax \
        _asm  mov   es,ax \
        }

#define EXIT_ISR() \
        { \
        _asm  mov   esp,ebp \
        _asm  pop   es \
        _asm  pop   ds \
        _asm  popad \
        _asm  iretd \
        }

__declspec(naked) void myisr(void)
{
        ENTER_ISR();
        /* your code here */
        EXIT_ISR();
}
```

Borland (32-bit) does not support C ISRs. It is necessary to write them in assembly. Note that a shell isr can be written in assembly that calls the actual handler, written in C, as shown:

```
DATA_SEL   EQU   14h      ;data selector assigned by pmEasy loader

enter_isr   MACRO
            pushad
            push ds
            push es
            cld
            mov  ax,DATA_SEL
            mov  ds,ax
            mov  es,ax
            ENDM

exit_isr    MACRO
            pop   es
            pop   ds
            popad
            iretd
            ENDM

            EXTRN     _myisrC:NEAR
            PUBLIC    _myisr
_myisr:     enter_isr
            call   _myisrC    ;call actual isr
            exit_isr


extern void myisr(void);  /* assembly shell */

void _cdecl myisrC(void)
{
      /* your code here */
}

void main(void)
{
      pmiSetPMIntVect(INT_NUM, myisr);  /* hook interrupt */
}
```

Underscores in names are necessary for interfacing with C, since the compiler prefixes names with one.

smx users: Use smx's ENTER_ISR & EXIT_ISR macros for C smx ISRs and enter_isr and exit_isr for assembly ISRs, as usual. 32-bit Microsoft users note that an additional pair has been created, EnterIsr and ExitIsr for non-smx-aware ISRs, to do the prolog and epilog (since the *interrupt* keyword is not supported). You may wish to write your own or put code inline that is as short as possible and only preserves the registers changed by the isr. (Reminder: an smx isr is one that enters the scheduler upon completion, usually to run an lsr invoked by the isr. Non-smx ISRs do not enter the scheduler, so it is not necessary to use the macros above.)

smx users should hook the isr or trap using the appropriate bsp functions. See XSMX\xbsp.h. Standalone pmEasy users should call the routines pmiSetPMIntVect() and pmiSetPMExcepVect() (setPMIntVect and setPMExcepVect for assembly), which use the DPMI server, to create an interrupt or trap gate and hook the isr. These DPMI routines create 16-bit gates for pmEasy16 and 32-bit gates for pmEasy32.

Final Notes: pmEasy, itself, uses 32-bit gates for its own trap and interrupt service routines. See the ISRs at the end of pme.asm and the BLDIDs near the top. Table 6-1 of

Intel's *386DX Programmer's Reference Manual* has a table of the various gate types. 16-bit gates say “286 ...” This is pointed out since most example interrupt descriptors in the rest of the book show the gate type set to a fixed value (a 32-bit gate) which misleadingly suggests that this is the only type of gate possible.

Saving and Restoring Interrupt Vectors

It is not necessary to save interrupt vectors prior to hooking them, unless your application depends on it. When pmEasy runs, the interrupt descriptor table (IDT) starts off as a blank slate. pmEasy hooks default handlers, but there is no problem with the application re-hooking these; there will be no problem returning to pmEasy.

There is also no problem returning to DOS, since real mode uses a completely different structure, the interrupt vector table (IVT), to store interrupt vectors. When pmEasy switches to protected mode, that table is no longer used, so it retains the state it had before pmEasy was started. When pmEasy returns to real mode, it sets the IDTR to point back to the IVT.

The only time it is necessary to save and restore a vector is if your application code needs to temporarily hook an interrupt level that it already hooked for another purpose. In that case, use pmiGetPMIntVect() and pmiSetPMIntVect(). Note that these functions get and save only the pointer size supported by the C compiler, so 16-bit applications save and restore a 16:16 pointer, and 32-bit applications store a 0:32 pointer. This works fine for normal use.

Things get a little tricky, though, if you step outside the bounds enforced by the C compiler. That is, if your code or a third party library uses assembly language, it is possible to set a 16:32 pointer. A call to pmiGetPMIntVect() in a 16-bit application would return a 16:16 pointer, which would lose the high word of the 32-bit offset. A call to pmiGetPMIntVect() in a 32-bit application would get only the offset and not the segment. Thus, it would not be possible to restore the original vector. In this case, the solution is to save and restore the entire interrupt descriptor (8 bytes) using pmiGetPMIntDescr() and restore it with pmiSetPMIntDescr(), so that you get the whole pointer.

Uninitialized Static Variables in C

C programs often assume that uninitialized variables are 0 until set. It is unnecessary to clear uninitialized data in the application startup code since this is done by pmEasy’s loader or the linker. For Microsoft 16-bit and 32-bit and Borland 32-bit applications, the segment or section specifies less data from the file than it is supposed to occupy in memory. The loader is supposed to (and does) clear anything beyond the data read from the file, up to the end of the segment or section. The Borland’s 16-bit linker seems not to take advantage of this trick and instead, it fills the .exe with 0’s where there is uninitialized data. This is unfortunate since the .exe ends up bigger than it has to be.

Flat Mode Segments for pmEasy32

The flat memory model offers simple treatment of the segment registers. CS, DS, and SS are loaded once for the application and never changed, and SS == DS. More precisely, the code produced by the C compiler never loads the segment registers. Assembly code can. Also, note that pmEasy changes the segment registers. Whenever a DPMI call (int 31h) is

made, pmEasy's DPMI server is entered and that code *does* change segment registers, but it restores them on return.

A file linked in the Portable Executable (PE) file format has no segment information. All fixups involve only offsets. This is because all code is referenced relative to the same selector value for CS, and the same is true regarding data access relative to DS.

Thus, only 2 descriptors are necessary for a flat mode application. Since we know this a priori, we can arbitrarily choose which ones to use. We use descriptors at 0Ch and 14h in the LDT to be code and data, respectively.

In making the call to the loaded application, pmEasy (in pme.asm) causes 0Ch to be loaded in CS. It is the responsibility of the application to set DS to 14h. See MINCAPP\start.asm.

Here is how relevant fields of the descriptors are initialized by pmEasy:

> base: 0
> limit: 4GB - 1 (0xFFFFF, with the granularity (G) bit set)
> D bit set (means use32 segment)
> G bit set (page granularity)
> Type bits are set appropriately for code or data

We advise the user to reduce the limits to the amount of physical memory actually present. This will trigger a general protection fault if a non-existent memory address is accessed.

Note that flat mode defeats the memory protection mechanisms built into the 386, unless paging is implemented. pmEasy currently does not support paging.

Tiled Descriptors for pmEasy16

pmEasy16 provides 16 descriptors in the GDT which map the first megabyte of memory. Each maps a 64KB block of memory. The first selector is tileSel0 and the others immediately follow. The first maps 0 to FFFFh; the second maps 10000h to 1FFFFh; etc. These are useful to access the first megabyte. They can be used in physical to logical address translation, and vice versa, as follows:

logical = MK_FP((short int)(((physical) >> 16) * 8) + tileSel0, (short int)(physical));

physical = ((dword)((FP_SEG(logical) – tileSel0) / 8) << 16) + FP_OFF(logical);

Debugging the Application

Debuggers Supported

pmEasy supports Soft-Scope® for debugging pmEasy applications. It is the only debugger currently supported for debugging EXEs. See the Soft-Scope section below for instructions on using it.

An alternative for smx users who use Paradigm C++ for application development is to build a located application for debugging (with the IDE's built-in debugger) and then build an EXE for release. Project files are supplied to build the smx Protosystem both ways.

See the release notes for any information about support for new debuggers.

Soft-Scope

Soft-Scope® is a remote debugger, meaning that it is divided into two parts: a user interface, which runs on the host, and a "monitor" that runs on the target. The monitor communicates with the host, returning the contents of registers, memory, and other data. The target monitor for Soft-Scope is CSi-Mon. It is linked to the debug (DBG) version of pmEasy. You must have our modified version of the CSi-Mon source code, which is supplied in PME\CSIMON when Soft-Scope is purchased from us. Do not use the CSi-Mon code supplied with Soft-Scope.

Connect the Target Computer

The target must be a 386 PC, or better, preferably running DOS, just as when running the release version of pmEasy. For the host, use your development system, running Windows 9x/ME/NT/2000/XP with Soft-Scope installed. Interconnect the systems with a null modem cable:

9 pin	25 pin	
2—3	2—3	
3—2	3—2	(Null Modem Cable)
5—5	4—5	
7—8	5—4	
8—7	7—7	

These are commonly available in computer stores for programs such as INTERLNK (DOS 6).

COM Ports and Speed

The target monitor is initially configured to use COM1, IRQ4, 57600 baud. To change the port or speed, you must alter the CSi-Mon source code.

To change the settings, you can either run CSICFG.EXE or make changes directly to CSICFG.INC and CSICFG.H. If you use CSICFG.EXE, you must re-make a change to CSICFG.INC, manually. Search for “USER:” in CSICFG.INC, prior to running CSICFG.EXE, and note the change. It is easier to edit the header files instead of using this utility for simple changes like this.

Change:

ADR_16450	to	3F8, 2F8, 3E8, or 2E8 (COM1, 2, 3, or 4)
BAUDRATE	to	115200, 57600, etc.
SI_INT	to	0x40 + IRQ (e.g. COM1 uses IRQ4, so 0x44)
SI_IRQ	to	IRQ (e.g. 4)

Run BUILD.BAT to rebuild CSIMON.LIB. Then rebuild pmEasy.

Note: Unmask the debugger interrupt in your application so that the Stop button will work. See DEBUGGER_IRQ_RX and _TX in bsp.h in SMX v3.6 and later. If you are using a non-standard COM port, you may need to adjust the code where it sets the baud divisor or change other initialization.

Copy to the Target

Copy the debug version of pmEasy (PME32D.EXE or PME16D.EXE) and the application (APP.EXE) to the target PC using a floppy disk or one of the other methods described in the subsection *Transferring the Application to the Target PC* of the section *Running pmEasy from DOS*, earlier in this manual.

Using CSi-Locate to Prepare Debug Info

Note: CSi-Locate had been periodically fixed to eliminate warnings caused by changes the compiler vendors make to the debug symbolics as they released new versions of their compilers. This tool is no longer under development, and the final version is 2.03.17. Ensure you have this version.

Note for pmEasy v1 users: Because of the enhancements made to the NE and PE loaders, the base selector (NE) or base address (PE) of the loaded application will be different than in pmEasy v1. In order for debug symbolics to match the code, the new APP16.CMD or APP32.CMD (in PME\CSIMON\CMD) must be used with CSi-Locate since it specifies the new base selector/address. pmEasy32 users who change HEAP_START must adjust the address in APP32.CMD accordingly.

Typically, CSi-Locate is used to create an OMF-386 (.abs) file with both the program and debug information. With that method, Soft-Scope downloads the code to the target for debugging and loads the symbolics into memory on the host.

Debugging in the pmEasy environment is different. Since pmEasy loads the application, the .abs file produced is used solely for the debug symbolics and the code in it is ignored. Fortunately, Soft-Scope provides a command to load just the symbols from the .abs file: **File | Symbol load**.

The locator requires a command file (.cmd) which tells it how to create the .abs file. Select the appropriate .cmd file we provide in PME\CSIMON\CMD, as indicated in the comment at the top of each. (The appropriate .cmd file is selected in the Protosystem makefile, so smx users need not worry — the Protosystem makefile uses this .cmd file and invokes CSi-Locate, if the **csiloc** macro is set to 1, performing steps 1 and 2 below, automatically.)

The .cmd file is quite simple — much simpler than it would be if the goal were to produce located code. The few commands present are needed just to fixup the debug symbolics to match where pmEasy loads the .exe.

1. Copy our .cmd file to the directory where your application's .exe and .map are located. (Both are inputs to the locator.)
2. Run CSi-Locate with the .cmd file as an argument. Example:

   csiloc app32.cmd
3. Any errors or warnings are reported to the screen. It is normal to get some warnings. Don't worry unless there are problems debugging or if there are many warnings. Past versions of the locator complained about changes to the debug symbolics made in newer versions of the compilers, and it was common to get many screenfuls of a few warnings. As the compilers change, we work with the developer of CSi-Locate to eliminate the warnings.
4. When done, an .abs file and a .cm file are produced. The .abs file is the all-important file containing the debug symbolics, and the .cm file is a log of what happened.
5. Assuming the locate was successful, you are ready to debug. Continue with the following section.

Soft-Scope Debugging Steps

1. Run PME32D.EXE or PME16D.EXE on the target. You will hear pmEasy loading the application.
2. While it is loading, start Soft-Scope on the host.
3. In Soft-Scope, load symbols with the **File | Symbol load** command in the menu. Specify the .abs file you produced with CSi-Locate.
4. When the application is done loading, a "PRESS ENTER..." message appears on the target. Press Enter> on target. This should cause the code window to pop up on the host, showing source code. This message will not be displayed if KEYBOARD is 0 in pme.inc. In that case, to pop up the code window, do Code | Display and then press Enter.
5. Buttons in the toolbar can be used for stepping through the code. Note that there are keyboard shortcuts for many commands. For example, "o" for step Over, "i" for step Into, etc. This is quicker and easier than the mouse for quick repetition. The shortcuts are specific to the currently selected window. For example, "o" and "i" only have this behavior if the code window is the currently selected window. These are apparently not documented in the current manual or on-line help but at least these two shortcuts mentioned do work.

In step 1, if you run app.exe which has the bound version of pmEasy using DISKRD21 as its disk reader, please wait until the application load completes before starting Soft-Scope on the host. Otherwise, if you are loading from floppy or other slow media,

Soft-Scope may attempt to connect while pmEasy has switched to real mode (via DPMI 300h) to run a DOS file i/o call for the load. Soft-Scope will complain that it gets no response from the target. Even if you try again when the load completes, it will still fail to connect. The solution is to exit pmEasy, run it again, and wait until the load completes before starting Soft-Scope.

Restarting the Application

With pmEasy v1, it was necessary to reboot the target between each debug cycle. This slowed debugging since PCs typically take awhile to reboot. pmEasy v2 has the ability to restart the application right from the debugger. Ensure RESTART is set to 1 in PME.INC.

1. Choose File | Restart from the menu. (File | Load does the same.) The "File name" line will have the name of the .abs file you loaded originally for the symbols. Leave it alone and leave the other two lines blank. Press OK. This restarts pmEasy and the application. This is what happens: CSi-Mon jumps to pmRestart (in PME.ASM), which switches to real mode and then jumps to pmInit, the entry point for pmEasy.

2. During the load, an error dialog will pop up "CSiMon — Receiver timeout". Ignore and dismiss it.

3. Press Enter on the target when the load completes

4. File | Symbol load, to re-load the symbolic information.

Please pardon the error dialog and the need to reload the symbols. Soft-Scope does not have built in support for pmEasy. Instead, we have done our best to integrate with the debugger as is.

This is a reliable way to do a restart, since the application is completely reloaded, restoring data and code to their original state. Also, pmEasy's data structures are re-initialized. Many, such as the descriptor tables, are initialized in the pmEasy code. The only thing that required special handling is the initialized data — the variables in pmEasy that are statically initialized. Because these are a bit scattered among the pmEasy assembly code, disk readers, drivers, and debug monitor, we decided that rather than changing all of these components to dynamically initialize their data, it would be easier to save the data at startup and then restore it on restart. Most of the _DATA segment is 0's, so it is compressed into a small space by scanning each word and saving only those that are non-zero. For each, we store the offset and its value. Then, during restart, the _DATA segment is cleared, then this table is processed, setting data to their original values. As long as the pmEasy code and the saved data image have not been corrupted, restart will succeed.

Soft-Scope Tips

1. Use the **src.path** macro in SSWIN32.INI to set the list of paths for Soft-Scope to search for source modules. They are searched in order. Normally, debuggers get the path from the debug symbolics, but since our makefiles use relative paths (..\..\xxxx) this does not work unless the current directory happens to be the one from which the .obj modules were built. Libraries for the different components of the SMX RTOS are built from different locations, so there is no single directory from which all files are built. Syntax is:

   src.path=c:\smx\protopm*.*;c:\smx\xsmx*.*;

The paths should be separated by semicolons and put in the order in which the directories should be searched. The *.* is necessary. Do not exceed the maximum line length (about 128 characters) or Soft-Scope will truncate your sswin32.ini file! You should save a backup copy of it.

2. If Soft-Scope cannot find a source file, it prompts you for the path. Once you type the path, Soft-Scope remembers and never asks again. Unfortunately, if you just press <Enter> without specifying a path Soft-Scope thinks there is no source for that module and there seems to be no way to change that during a session. The only solution seems to be to exit the debugger and delete the .tmp file (e.g. app.tmp) produced by Soft-Scope. Then restart Soft-Scope. Soft-Scope creates this file to save symbolic information for the application you are debugging so that loading it is much faster for your next session. The file is only used if you have not made changes to the app you are debugging.
3. Baud rate: Using a fast baud rate might actually make Soft-Scope response slower, if there are many communication retries at the higher baud rate. You may find that higher rates offer no improvement or even make things worse. You can determine how many communication retries are occurring by generating log files (see tip 4, below) while running at different baud rates and comparing results.
4. If you are curious to see what is happening in debugger-to-monitor communication, you can produce a log file by adding **targ.debug=filename** to the SSWIN32.INI file. If no path is specified, the file is created in the Soft-Scope directory. To direct it to the printer instead, use **targ.debug=prn**.
5. If you step to the next instruction and the Soft-Scope's pointer symbol goes to the end of the file, this is apparently normal. It occurs when the current instruction is the last one in the file. This is probably just a characteristic of how Soft-Scope handles line number information. If you step, the result is the same as if the pointer had been on the proper line.
6. smxAware offers smx-awareness to Soft-Scope. The commands to add to SSWIN32.INI to enable smxAware are documented in the smxAware User’s Guide.
7. The comment symbol for the SSWIN32.INI file is a semicolon. This makes it possible to store several src.path macros, for example, so it is easy to switch to another set of paths to debug a different project.
8. In order to be able to stop the application with the Stop button, your application needs to unmask the IRQ used by the COM port used for the debug link. For SMX, this is done by setting DEBUGGER_IRQ_RX and _TX in bsp.h (in SMX v3.6 and later).

Soft-Scope Limitations

Problems reported in earlier versions of this manual have been mostly solved by newer versions of CSi-Locate. Of course, the compilers continue to change, so there could be new problems in the future. Please consult the release notes if there is trouble, and contact us if that doesn’t help.

Techniques If Not Using a Debugger

Locating a GPF or Other Exception

The easiest way to locate and determine the cause of a processor exception is using the debugger. When an exception occurs, the debugger will stop on the instruction that caused it. The registers and stack reflect the conditions at the time the exception occurred. Often it is fairly clear what the problem was, such as loading a segment register with an invalid selector or exceeding a segment limit. However, you may need to dig a bit more to find out where the invalid selector value came from or why the offset exceeded the segment limit. Since these debuggers show source code, you should be able to determine fairly quickly where the problem is. Knowing that, in your next debugging session, you can stop the debugger before it and step through the code leading to the problem.

If you do not have a debugger supported by pmEasy, you can still locate the cause of an exception, but with a little more effort. pmEasy has default exception handlers which report when a processor exception occurs and display the address of the faulting instruction. An error code is additionally displayed for some exceptions. (Whether the address of the faulting instruction and whether an error code is displayed is specified in tables 9-5 and 9-6 in the Intel *386DX Programmer's Reference Manual*.)

From the error message displayed by pmEasy's exception handlers and the .map file, you can determine the routine where the exception occurred, or the closest public routine before it. Once you have found this routine, generate an assembly listing file (.cod or .lst) for the module the routine is in, with the compiler or assembler. Then, using the offsets at the left, determine which statement caused the fault. Specifically,

1. Determine which public routine is closest before the fault location. Look at the part of the .map that shows symbols in order by value.
   a. For segmented applications, determine which segment it occurred in, from the address displayed by the handler. Note that the segments in the .map are numbered 1,2,3, etc, while the actual selectors may be 14h, 1Ch, 24h, etc. The order of the segments in the .map is the same as the order they are loaded by the pmEasy loader, as displayed in the “LDT Map” during loading.
   b. Looking at the symbols in the appropriate segment or section, find the one with offset closest to the address of the fault (before it). This is your reference point. Most likely the fault occurred in this routine. However, since static functions are not public, they do not appear in the .map, so it may be in a function following the one in the .map, but this does not usually matter.
2. Subtract the offset of the closest symbol in the .map from the offset reported by the exception handler. This is the location where the exception occurred, relative to the start of that routine.
3. Look at the .cod or .lst file for the module with your editor. These files show offsets (relative to the beginning of the file) for each instruction. Go to the start of the routine found in step 1b. Add the relative offset from step 2 to the offset shown in the file. That will give you the offset in the file where to look. If this is not the start of an instruction, ensure that your calculator was set for hexadecimal for the entire calculation. If that was not the problem, ensure that none of your files is out of date. It may be wise to re-build your program clean.

Now that you have found where the exception is occurring, you still need to find the cause. This will require desk-checking your code.

Although feasible, this technique is time-consuming. It is probably worth the relatively small investment to purchase a debugger!

pmEasy Operation Summary

Please read this chapter before modifying pmEasy. It gives an overview of how pmEasy works.

Entering Protected Mode

pmEasy's entry point is *pmInit* in pmi.asm. The segment registers are first loaded appropriately. Then _BSS and c_common are cleared. These are the uninitialized data segments. Clearing them is normally done by the C startup code, but that is not present for pmEasy. Thus, pmEasy clears these segments to uphold the C convention of setting uninitialized variables to 0.

The interrupt mask is saved (for exit from pmEasy to DOS) and all interrupts are masked.

The screen is cleared and the pmEasy banner is displayed, followed by a check for the presence of another DPMI server. If one is present, execution is aborted. (See *Running pmEasy from DOS* in the chapter *Running pmEasy*.)

If RESTART is enabled, the _DATA segment is copied to another location in RAM so that when pmEasy is restarted the _DATA segment can be restored to its initial values and pmEasy will restart properly. pmEasy is restarted when the application is restarted in the debugger. See the Soft-Scope section of the chapter *Debugging the Application* for more information.

Address line A20 is enabled to allow accessing high memory locations. It is disabled upon return to DOS.

pmEasy then determines the name of the application to load. It first checks to see if the application name was passed on the command line. If so, the default drive letter and extension are added to the name, if missing. If no name is passed on the command line, pmEasy checks the environment block of the PSP to see what was the name of the program run from the DOS prompt. If the name starts with something other than "pme" pmEasy assumes it is the name of the application and that pmEasy is bound to it. In this case, the name is copied to the file name buffer. Otherwise the default application name, *appName*, is used.

Next is a series of calls to the BLDSD macro which initializes descriptors in the global descriptor table (GDT) . These calls set up the segments used by pmEasy, setting their bases and limits as well as their access types. The IDT is statically initialized to a default isr in all descriptors. Individual vectors are later hooked (see below).

The GDTR and IDTR registers are then loaded. These registers indicate the location and size of the global descriptor table (GDT) and interrupt descriptor table (IDT), respectively. Each of these registers is a 6-byte register: 2 bytes for the limit followed by 4 for the base address. The base address is a linear address, as is true of all descriptors. Loading these is a two step process. First the values are copied into 6-byte variables and then the LGDT and

LIDT instructions are issued to copy from those to the actual registers in the processor. Figure 4-2 of Intel's *386DX Microprocessor Programmer's Reference Manual* shows these registers. Before changing the IDTR, though, its current contents are saved in *idtsave* so the original value can be restored upon exit to DOS. SS and SP are also preserved.

A call is then made to the *picsPM* routine which reconfigures the peripheral interrupt controllers (PICs) so that the interrupt levels are changed. The master PIC is changed to generate interrupts 40h to 47h, and the slave PIC is changed to generate interrupts 48h to 4Fh. This moves the master PIC interrupts so they do not conflict with protected mode exception interrupt levels (it normally has range 08h to 0Fh). On exit from pmEasy, the PICs are restored to their original state. After *picsPM*, interrupts are still disabled and then re-masked so they can be unmasked individually.

It is now time to switch to protected mode. This is probably the simplest step, requiring only that bit 0 (the PE bit) of the CR0 register be set to 1. Now that the processor is in protected mode, we jump to a new routine, *pmMain*, in PME.ASM. Note that a JMP16 macro is used rather than a JMP instruction. This is necessary because a real mode far JMP or CALL uses segment addresses, not selectors, which is wrong for protected mode. (Keep in mind that pmEasy is loaded as a real mode program.)

Note that in your application code, it is ok to use normal jumps and calls because it will be linked as a protected mode application and will properly get selectors rather than segments for far addresses.

Basically, the JMP16 macro actually constructs a jump by supplying the opcode for the jump instruction, followed by the selector and offset to jump to. Ensuring that selectors are used rather than segments for far addresses was one of the difficult parts of developing pmEasy since it had to be done everywhere, even in C (many MK_FPs are used to “manually” construct far addresses).

Running in Protected Mode

pmMain starts by reloading the segment registers with selectors, to replace the real mode segments that are currently in them. The selectors correspond to descriptors in the GDT created by the BLDSD macros, as mentioned above.

The stack is cleared in order to determine stack usage of the loader, disk reader, and driver. The usage percentage is displayed to the screen after the load completes.

The extended part of the general purpose registers are cleared as a precaution (the upper 16 bits of eax, ebx, etc.).

Next the IDT is initialized by calling *hookVects*. Prior to this step, it had all entries set to the default isr, *genIsr*, which does nothing but return. Now particular levels are loaded with the addresses of actual interrupt and exception handlers, using the BLDID macro. These handlers are defined near the end of PME.ASM.

The PC tick is set to the standard rate of 18.2 ticks/sec, in case it wasn’t already set to that value, since some delays in pmEasy assume this.

The LDTR and TR registers (similar to GDTR and IDTR) are then loaded. These are simpler to load than the other two, since only a selector needs to be loaded. The LDT and TSS0 are actually segments in themselves (which were created in PMI.ASM with the

BLDSD macro), and each has a descriptor in the GDT. When the selector is loaded into LDTR or TR, the processor automatically reads the corresponding descriptor of the GDT. (Remember that these selectors are indexes into the GDT.)

dpmiInit is called to initialize the DPMI server. Mostly this routine sets up the heap. The heap is where the application will be loaded and where temporary blocks used during the load are allocated.

If DEBUGVER is defined (it is defined in the makefile for the debug (“d”) version of pmEasy), the initialization routine for the Soft-Scope monitor is now called. (The monitor is linked with pmEasy and communicates across the serial link to Soft-Scope running on the host.)

Loading, Running, and Exiting the Application

At this point, the application loader is called. The loader allocates space for the EXE image in the heap, copies it from disk, and performs fixups. It returns the entry point of the loaded application in a LD_INFO_BLK structure. Stack usage is checked (the stack had been cleared at startup) and displayed on the screen as a percentage. This allows you to determine if there was sufficient stack for the loader, disk reader, and driver. Nothing else in pmEasy uses much stack, so if the load succeeds without overflowing the stack, the stack is big enough.

Execution waits at *waitStart* or *waitHost* for the user to press Enter. *waitStart* is used in the release version so the user can see the output from the loader. This is only intended for use during development. When ready for release, set KEYBOARD to 0 in pme.inc.

waitHost is used in the debug version to synchronize with the debugger. It is not absolutely necessary, but by running the debugger before the following code runs, it causes the code window to automatically pop up. Otherwise it has to be opened manually. This is accomplished by the following lines which set the processor’s Trap Flag. This causes a breakpoint interrupt on the instruction after the next. Since the next instruction is the call to the application, the result is that the debugger is stopped on the first instruction of the application. (See the Soft-Scope section of the chapter *Debugging the Application* for detailed instructions on running this debugger.)

The call to start the application is done through the entry_pt_seg and entry_pt_off fields of the appInfo (type LD_INFO_BLK) structure.

If the application saves the stack pointer in its startup code (as the smx Protosystem and the pmEasy minimal applications do, it can exit to pmEasy. It simply returns to the next instruction after the call. Prior to exiting, the application can set AX to an exit code, which pmEasy then displays on the screen. This feature is for your use. You may remove it if you choose.

The pmEasy banner is restored. The pmEasy interrupt vectors are re-hooked (e.g. tick and keyboard ISRs). unload() is called (in the loader module) to free any resources (memory and descriptors) used by the application. This is optional and useful only if pmEasy is modified to load another application. There is no need to do this if exiting to DOS or rebooting, since DOS is unaware of the pmEasy heap or descriptor tables.

If exiting to DOS, the PS/2 mouse is disabled in case the application was using one and did not disable it before exit. This is necessary because both are controlled by the same 8042

controller and any mouse movement in DOS would cause the keyboard to stop working. The keyboard is also reset, to ensure it will work.

Pressing Esc at this point causes pmEasy to exit to DOS or reboot.

Exiting pmEasy

The first thing done to exit to real mode is to load the segment registers with the selectors of descriptors that have appropriate values for real mode. Most importantly, the limits must be FFFFh. The base addresses are set to DGROUP. Other bits in the descriptor must have specific values. (For discussion of these values, see p. 9-3 of the *386 System Software Writer's Guide*. p. 10-3 of *386DX Programmer's Reference Manual*, which shows the initial values of the registers after system reset, may also be helpful.) If this is not done, you may see something such as “internal stack fault, system halted” when exiting.

Loading the segment registers with such descriptors is necessary because even in real mode, the 386 uses the hidden part of the segment registers (the “pseudodescriptor”) . It adds the offset to the base address field in constructing addresses and checks the limit field for address validity.

Loading CS is accomplished with a far jump. Interestingly, it is necessary to do this prior to returning to real mode, since it is only possible to load the hidden parts of the registers when in protected mode (except the base field, which is set to (seg * 10h) every time a real mode segment is loaded into the segment register. This allows the 386 to mimic 8086 addressing).

The PE bit of CR0 is then cleared to leave protected mode. Execution returns to PMI.ASM, in a far jump to *rmReturn*. *rmReturn* reloads the segment registers with segments rather than selectors and then reboots, if not returning to DOS.

If returning to DOS, the PICs are reprogrammed back to their original interrupt levels, and the original IDT is restored by loading IDTR with *idtsave*. The DOS sector cache is cleared in case the application altered any disks in the system (see *Clearing the DOS Sector Cache* below). A20 is disabled (DOS will automatically re-enable it if DOS had it enabled before running pmEasy). The screen is restored to text mode using BIOS int 10h function 0h, in case the application had switched the screen to graphics mode. Note that any messages pmEasy displayed on the screen after exiting the application will now appear. This works because graphics and text video memory are separate and the text writes were written to text video memory even though the video controller was in graphics mode. (Since this is a BIOS call, it must be done in real mode, so it couldn’t be done earlier when the application exited.)

Note that some DOS programs enable address line A20 in order to use the 64K-16 bytes above 1MB. By default, pmEasy's heap begins at 1M+64K-16 bytes (0x10FFF0), to avoid overwriting memory which may be used by such a program. Keep this in mind if you change *HEAP_START*, since pmEasy may not successfully exit to DOS if you allow this high 64KB to be overwritten. Starting pmEasy’s heap at 0x10FFF0 causes the application image to start at 0x110000 since the first 10h bytes are occupied by the heap control block.

Clearing the DOS Sector Cache

On exit to DOS, pmEasy runs a routine to clear the DOS sector cache. This is important if the application makes any changes to the filesystem (such as with smxFile). These sector buffers are the ones DOS allocates by the BUFFERS= statement in config.sys (it defaults to 15 even if not specified). DOS stores recently accessed sectors in these blocks, primarily FAT and directory sectors. If the cache is not cleared, then on return to DOS, it has an outdated view of the contents of the disk. A likely result is cross-linked files if you copy new files to the disk without rebooting. Now it is safe to use the disks on exit, except for DiskOnChip.

You must reboot before accessing a DiskOnChip if the application modified it, since M-Systems's driver for DOS apparently caches its own information and we currently do not know whether it is possible to flush it or reset the driver. As a result, on exit from an application that modifies the DiskOnChip, the root directory will appear scrambled until you reboot. We hope to solve this problem in a future release.

pmEasy Internals

It is not necessary to read this chapter in order to use pmEasy. However, several topics are discussed that will be helpful for those who wish to modify pmEasy or are curious about pmEasy's internals.

Heap Structure and Initialization

pmEasy implements a simple heap for memory management. Each block of the heap is preceded by a heap control block (hcb). By default, the heap initially starts as a single free block. It is preceded by an hcb and the end is marked by a second hcb. As blocks are allocated, the space is divided. A simple, first-found scan is used. The standard function (pmiAllocMem() / DPMI 501h) searches starting from the lowest address of the heap. A non-standard function (pmiAllocMemTop() / DPMI 581h) is provided to scan from the top down. The latter is intended to be used only in special circumstances — see the descriptions of these functions in the *DPMI/CPMI Reference*, earlier.

Each time a block is allocated, a new hcb is allocated (unless it fits exactly into a hole left by a previous block that was freed). The new hcb marks the start of the remaining free memory, indicating it as available. When blocks are freed, they are merged with adjacent free blocks.

The HCBs are 16-byte structures consisting of 4 fields:

heap control block (hcb)

0x55555555		bfence
flnk	000 i	
blnk	0000	
0x55555555		efence

The *flnk* and *blnk* fields serve to create a doubly-linked list of blocks in the heap to allow the heap functions to scan for free blocks. The *fence* fields are used as a way to detect heap corruption. They are set to a constant pattern, so it is evident if one has been overwritten.

In-use bit (*i*) : Because HCBs are aligned on paragraph (16 byte) boundaries, the low 4 bits of the *flnk* and *blnk* pointers are not needed to address the next block. These are 8 bits we can use for special purposes. Currently, we use only one of these bits: the low bit of *flnk* is used to specify whether the heap block that follows is in use (1) or available (0). You will notice from study of the 5xx routines in DPMI.ASM that we mask out the low bits of the *flnk* pointer before using it as a pointer to traverse the linked list of HCBs. Using this trick makes a small amount more work for the heap routines, but it minimizes overhead of the HCBs by keeping them small. 16 bytes is a convenient size.

Heap Regions

The heap can span across gaps in memory. For example, it can start below 640K up to that point and then continue in extended memory. Or it can be used to avoid memory-mapped I/O space, if any. This is accomplished by setting HEAP_START and the *heapRegions* table appropriately. Each line of the table specifies a region of RAM to include in the heap. Heap initialization automatically excludes the gaps by writing an HCB before each and marking it as a block that is in use.

Auto-Sizing

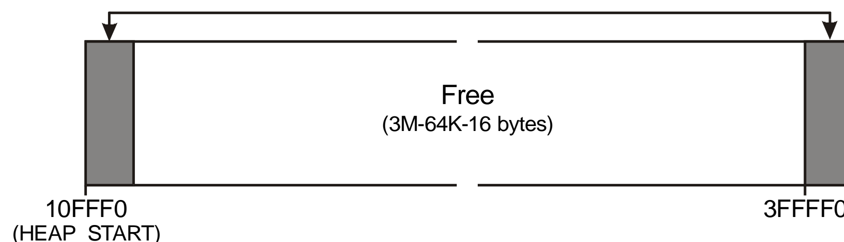
Another nice feature of pmEasy heap initialization is the ability to scan for the end of RAM and set the end of heap there. This is done if the ending address of the last entry in the *heapRegions* table is 0. The scan is done at user-defined increments (SCAN_INCR in pme.inc). The last HCB is written at the last 16-bytes of memory.

This feature does not work reliably in all systems. We have found that some PCs apparently have I/O space at the top of extended memory, and the probing done by the scan can cause the system to hang, if it writes to a sensitive I/O location. We experienced this problem on the IBM E-Pro, which appears to have I/O space in the top 512KB of the memory space. It would hang on the next tick interrupt after the probe. It does not solve the problem to save what is at the location and restore it after the probe because the problem is writing to an I/O register, not just the need to preserve a memory location. The symptom is that the target will hang after the HEAP INIT message is displayed but before pmEasy's loader runs. If you observe this, you will have to explicitly set the end of heap (below the I/O space). Initially try setting a very small heap to quickly see if this solves the problem.

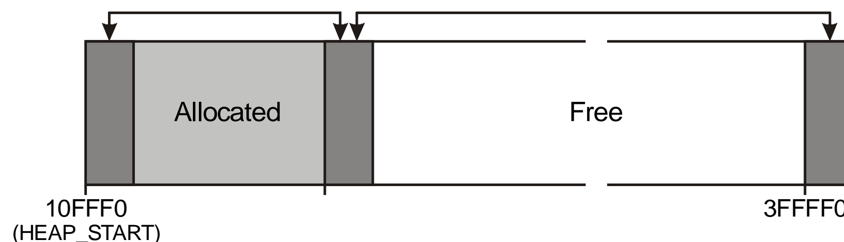
Heap Diagrams

Note: In these diagrams, double-headed arrows denote a doubly-linked list. All pointers point to the first byte of the HCB. Addresses shown are examples. The beginning and ending addresses may be anywhere.

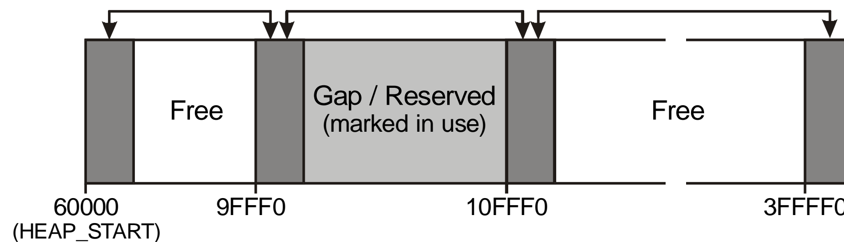
Simple Heap at Startup



Heap after One Allocation



Heap with Multiple Regions at Startup



As the diagrams show, creating a heap with gaps is just like a heap with allocated blocks.

Buffer Area

pmEasy defines an area of memory below 1MB as the Buffer Area. Its starting and ending addresses are configured by the BUFFER_AREA_BEGIN and BUFFER_AREA_END constants in PME.INC and PME.H. This area is currently used for two buffers. More may be added in the future, if needed. The two buffers currently used are the floppy driver DMA buffer and a buffer for saving a copy of the _DATA segment.

The floppy DMA buffer must be located in memory below 1MB at an address such that it does not cross a physical 64KB boundary. Because of this, it cannot be allocated statically in the pmEasy image since there is no guarantee where DOS will load pmEasy. That depends on what drivers and TSRs are loaded at startup. Also, it is preferable not to allocate large buffers in pmEasy, since that just makes the .EXE bigger and might prevent converting it to a binary file (PME.SYS) with EXE2BIN. For these reasons, we have defined the Buffer Area. Please see the Buffer Area settings in the chapter Configuration Reference for discussion of the settings for the DMA buffer and _DATA buffer.

By default, the Buffer Area starts at linear address 90000h. We tried setting it to 98000h, but then pmEasy would no longer return to DOS, so we presume DOS or the BIOS is using some portion of the memory between 98000h and A0000h. If you have problems exiting to DOS, try reducing this value. Rebuild the disk reader and pmEasy, if you do.

If memory is tight and you want to add the unused memory below 1MB to pmEasy's heap, be sure to avoid the Buffer Area. Do this by setting the first region to end just below it and the next to start in extended memory. This way, the Buffer Area is part of the gap in the heap for the PC ROM area.

PIC Configuration

The BIOS on a PC configures the Programmable Interrupt Controllers so they generate IRQ0 to IRQ7 on interrupts 08h to 0Fh and IRQ8 to IRQ15 on interrupts 70h to 77h. This is fine for real mode, but it is a problem for protected mode because interrupts 08h to 0Fh are processor faults. If IBM had heeded Intel's documentation marking these low interrupts as Reserved, we would have been spared this extra complexity. The solution is to reprogram at least the master PIC to generate higher interrupt levels. It is common to reprogram the master PIC to generate interrupts 78h to 7Fh. The problem with this, though, is that it is non-intuitive — the PICs are in "reverse order" (78h to 7Fh, 70h to 77h).

We felt it was simplest to reprogram both PICs, so that IRQ0 to IRQ15 generate interrupts 40h to 4Fh. pmEasy does this in *picsPM*. On exit, *picsRM* is called to put them back to their original configuration for return to DOS.

It is a simple matter to change *picsPM* if you prefer to configure the PICs to generate different interrupts. Those using only pmEasy may do this if they alter the disk reader to hook the right interrupt levels for the disk ISRs. We do not recommend that smx users do this, but it can be done if: (1) a complete search for pmiSetPMIntVect() is performed in all smx modules, and the code is altered to use the correct levels, and (2) if the debug monitor is altered accordingly.

Small Model Only

pmEasy must be built for Small memory model in order to minimize use of far pointers. Since pmEasy is built with the 16-bit compiler and linked as a real mode .exe program, the linker creates all far addresses using segments not selectors. If built for any of the other memory models, code and/or data would be put into separate segments for each module, so that references from one to another would necessitate far pointers.

Inspecting the pmEasy .map file shows that all code is in the _TEXT segment. Similarly, all data segments are grouped into DGROUP (see the GROUP statement in PMI.ASM; this is not shown in the .map). Thus, there is only one segment for each. If there were multiple code segments, calls between routines in different segments would require manually forcing use of selectors rather than segments, as is done in the CALL16 macro. Likewise, far pointers in C would have to be manually constructed for all inter-segment data references, as is currently done for some pointers in the loaders.

SS == DS

Notice that there is no separate STACK segment. The stack is in the data segment and the compiler switch is used to ensure small model with SS == DS. Some C routines in the loader, disk reader, and drivers take pointers as parameters. If the caller defines an autovariable (a non-static local variable), and then passes it to such a routine, there is a problem. The actual variable whose address was passed is in the stack so its address would be relative to SS, but the called routine treats it like any near pointer and assumes it is relative to DS. This works fine as long as SS == DS. Otherwise it is a problem. The usual way to circumvent this problem is to define the parameter as a far pointer so that the selector is passed and not assumed, but this does not work for pmEasy because, it is linked as a real mode program, so a segment value is used rather than a selector. Do not try to define a separate stack segment in pmEasy.

Constructing Far Pointers

The loaders need to be able to allocate memory for the application from pmEasy's heap and write code and data there. This requires use of far pointers. The loaders use MK_FARP(a), which is defined as MK_FP(dataSel, a), where *a* is a near address. This is the technique that should be used whenever a far pointer is truly needed.

Altering pmEasy

Before modifying pmEasy, please see the chapter *Configuration Reference* to see if an option is already provided to do what you need. For example, there are settings to disable video output and the requirement to press a key to start the application running when the load completes. There are many other options.

All source code for pmEasy is provided except for the drivers used by the disk reader. Driver source code can be purchased, or it is provided if the corresponding drivers are purchased for smxFile. The CD-ROM reader is smxCD.

If you need to add features, you can write code in C or assembly and easily add new modules to the makefile using the modules there as a guide. Before making any changes, first verify that you can successfully build pmEasy and run it. If not, you may be using the wrong tools. See *Building* pmEasy in the *Introduction* chapter and the *Tools* chapter in the early part of this manual.

Please refer to the chapter, pmEasy *Internals*, for information that is relevant for making modifications to pmEasy.

Far Addresses

Note when modifying pmEasy that some tricks are necessary due to the fact that it runs in protected mode, yet is linked and loaded as a real mode program. In particular, be careful to use selectors rather than segments in far calls in protected mode. You must “manually” create far addresses since the loader or locator supplies real mode segment values rather than selectors.

A trick to ensure that you have constructed all far addresses correctly, using selectors rather than segments, is to compare the number of segment fixups in the .exe before and after you add your code. The word at offset 06h in a DOS .exe specifies the number of entries in the relocation table (or see “Relocations” in the output of the **EXEHDR** utility provided with the Microsoft compiler). This number should remain the same after you add protected mode code. It is non-zero, since there is real mode code in pmEasy for entry to protected mode and exit, which makes segment references. Note that adding real mode code might cause the number to increase. It is ok to introduce new segment references in the real mode portions of the code.

Tip: If you have added code to the protected mode part of pmEasy and you cannot determine why it is not working, it is worthwhile to check all fixup locations to ensure none are in protected mode code. The Borland TDUMP utility shows the addresses of all relocations in the .exe. Redirect the output of this utility to a file (e.g. tdump pme32 > hdr) for study. Comparing this to the .map file and .lst/.cod files will show where the fixups are.

Write pmEasy ISRs in Assembly or with Shells

If you need to write an isr in pmEasy (like those used by the disk drivers), write them in assembly or write a shell in assembly that calls the main isr written in C. This is necessary because the *interrupt* keyword is not correct for protected mode, when the code is compiled and linked for real mode. The problem is that in the function prolog, the DS register is set with a segment rather than a selector value. In assembly, you can ensure the proper prolog and epilog in a shell isr that calls the C interrupt handling routine to do the work. Copy the enterIsr and exitIsr macros in LOAD\DISKRD\isr.asm for use in your shell isr.

Debugging pmEasy

Debugging pmEasy is difficult because it switches processor modes, and debuggers are suited to debugging programs in one mode but not both and certainly not during a transition between modes. This chapter discusses what can be done.

Real Mode Code

It is possible to debug the real mode portion of pmEasy using CodeView. There is a strange problem that causes it to initially display code not at the instruction point, but there is a workaround. Follow these directions if you need to debug:

1. Set the **debugpme** macro to 1 in PME.MAK and rebuild pmEasy clean. This compiles and links debug symbolics.
2. Type "**cv pme32.exe**" to load it into the debugger.
3. When CodeView prompts you for the location of a module, press **Esc** to clear the dialog. (The code in the source window will **not** be the code about to execute.)
4. Type "**g pmInit <Enter>**" in the command window. Then debugging should be fine.

Note that if at step 4 you typed . <Enter> this would cause the code window to display the true first instruction of the program, but then a single-step would cause the program to free-run, for some unknown reason. It is not yet clear to us what it is that CodeView doesn't like about pmEasy.

Protected Mode Code

It is possible to debug much of the protected mode code in pmEasy, including the loader, but unfortunately, only disassembly, not source-level debugging, is supported. There is a line in PME.ASM, "EARLYDEBUG". Uncommenting it will allow debugging from the next instruction after this line with Soft-Scope. You can use the /Fc switch to generate debug a mixed C/assembly listing file so you can track progress through the code. This is tedious, but certainly better than debugging solely with print statements.

The reason source level debugging is not supported is because the addresses in the debug symbolics use real mode segment values instead of selectors, so the debugger cannot correlate the symbolic information to the code. What is needed is a way to translate the addresses in the debug symbolics to use selectors. An older debugger we supported had a way to do this. Unfortunately, Soft-Scope does not.

Markers

You may have to resort to the old technique of sprinkling markers in the code. Enable the MARKER macro in pme.inc. It uses the serial port, which could have problems of its own, so you may wish to re-implement this macro to display a character to the screen or light LEDs, etc.

Configuration Reference

pmEasy can be configured and modified as necessary to suit your needs. We suggest that you initially use the pre-built pmEasy executables provided and start developing the application before changing the configuration settings or modifying source code.

Below is a cross-reference used to quickly locate settings by section.

Alphabetical Cross Reference of Configuration Settings

Setting	Section
A20 settings	A20 address line
appName	Loader and Loading
BAUD (Soft-Scope)	Debugger
BINDING_SUP	Binding to Application
BIOS	Target BIOS/DOS
BUFFER_AREA_BEGIN	Buffer Area
BUFFER_AREA_END	Buffer Area
cdfsdriver	Disk Reader Drivers
CHECKSTACK	Diagnostics
COM port (Soft-Scope)	Debugger
coproPresent	Diagnostics
csiloc	Running pmEasy from ROM
DATA_SAV_BUF_BEGIN	Buffer Area
DATA_SAV_MAX_SIZE	Buffer Area
debugpme	Diagnostics
DEFAULT_DRVXT	Loader and Loading
DefaultDrv	Loader and Loading
DefaultExt	Loader and Loading
DIAGNOSTIC_MSGS	Disk Reader
DISKRD	Disk Reader
DISKRD21	Disk Reader
DOS	Target BIOS/DOS
DPMI100 and DPMI300	DPMI Services
ENABLE_CDROM_CODE	Disk Reader
ENABLE_DISK_CODE	Disk Reader
FIRST_ATAPI_CDROM_UNIT	Disk Reader
FL_DMA_BUF_BEGIN	Disk Reader Drivers
FL_DMA_BUF_SZ	Disk Reader Drivers
floppydriver	Disk Reader Drivers
FNAME_SZ	Loader and Loading
GBAND_VAL	Heap (Protected Mode), and Stack
gdtEntries	Descriptor Tables

HEAP_START	Heap (Protected Mode)
heapRegions	Heap (Protected Mode)
HEXLD_BUFSIZE	Loader and Loading
HEXLOAD	Loader and Loading
idedriver	Disk Reader Drivers
idtEntries	Descriptor Tables
KEYBOARD	Keyboard
ldtEntries	Descriptor Tables
LOAD	Loader and Loading
loader	Loader and Loading
msysdriver	Disk Reader Drivers
PARSECMDLINE	Loader and Loading
PELOAD	Loader and Loading
PMSTACK_SZ	Stack
quietLoad	Loader and Loading
READBUFSZ	Loader and Loading
realHeapSize	Disk Reader Drivers
RESTART	Debugger
RSTACK_SZ	Stack
RTFS	Disk Reader
SCAN_INCR	Heap (Protected Mode)
SOFTSCOPE	Debugger
UNLOAD	Loader and Loading
VIDEO	Video Output
videoSeg	Video Output

Configuration Settings Reference

A20 address line:

A20 (pme.inc, pme.h)

This should be set to 1 so that the A20 address line is enabled, giving access to extended memory. It should be set to 0 only on systems for which the A20 line is already enabled. Also, at least one of A20_92 and A20_KBD must be set. If both are set, the code first tries to enable A20 via port 92h and then if that fails, it tries via the keyboard controller.

A20_92 (pme.inc, pme.h)

Enables code that sets A20 via port 92h (newer PCs).

A20_KBD (pme.inc, pme.h)

Enables code that sets A20 via the keyboard controller (older PCs).

Application Name: (see appName in the Loader and Loading settings below)

Binding to Application:

BINDING_SUP (pme.inc, pme.h)

Set to 1 to enable code necessary to properly support binding pmEasy to the application. See the section *Binding pmEasy to the Application* in the *Running pmEasy* chapter for more information.

Bootloading:

Several constants must be set to create a bootloadable version of pmEasy, and a bootable disk is created using utilities we provide. Please refer to *Bootloading pmEasy from Disk* in the *Running pmEasy* chapter for full instructions.

Buffer Area:

BUFFER_AREA_BEGIN and BUFFER_AREA_END (pme.inc, pme.h)

Set these to span an unused area of RAM that is large enough to hold the floppy driver DMA buffer and a copy of the _DATA segment. See the Disk Reader Drivers settings below for discussion of the floppy DMA buffer settings and size of the buffer. The _DATA segment is copied to the area starting at DATA_SAV_BUF_BEGIN when the RESTART option is enabled. This allows the debugger to restart pmEasy. It is necessary to save the initial state of the data since pmEasy changes some variables while it runs and these must be restored to their original values in order for pmEasy to restart.

These beginning of this area is at linear address 90000h, by default. We tried setting it to 98000h and found that pmEasy would not return to DOS. Apparently, DOS or the BIOS uses some portion of the memory just below A0000h.

Note: If HEAP_START and heapRegions are set to start the heap in real memory ensure the heap does not overlap the buffer area used for the floppy driver DMA buffer or the saved _DATA image. Set the ending HCB of the first region below the start of this area, so that this area is included in the gap for the PC ROM space.

If BUFFER_AREA_BEGIN is changed, DISKRD must be rebuilt.

DATA_SAV_BUF_BEGIN and DATA_SAV_MAX_SIZE (pme.inc)

Set these to addresses in the buffer area. Ensure they span a large enough block of memory to hold the _DATA segment (see the .map file). See the BUFFER_AREA settings above for more discussion.

FL_DMA_BUF_BEGIN and FL_DMA_BUF_SZ:

See the Disk Reader Driver settings below.

Debugger:

COM port, IRQ, and BAUD for CSIMON (Soft-Scope) (CSIMON\CSICFG.H and .INC)

The CSICFG.EXE utility could be used to make the change, but we suggest instead that you directly edit these include files, since an important change is lost every time these are re-generated. This is explained more fully in the readme there.

RESTART (pme.inc)

Set this to 1 to enable application restart in the debugger. This actually causes pmEasy to restart and reload the application from disk. This is a good way to restart, since pmEasy is re-initialized and the application is reloaded and initialized. We recommend that you copy the application to the target's hard disk (if it has one) for fastest reloads. This option is enabled only for the debug version, by default. If you have a use for this in the release version, enable RESTART_PME in pme.inc.

SOFTSCOPE (pme.inc)

Select the Soft-Scope debugger by setting this to 1.

Descriptor Tables:

gdtEntries (pme.inc)

Specifies the number of descriptors in the GDT. Typically, there is no need to change this.

idtEntries (pme.inc)

Specifies the number of descriptors in the IDT. Typically, there is no need to change this, unless memory is tight. 256 is the maximum number of interrupt vectors possible. This table could be reduced to the highest vector used.

ldtEntries (pme.inc)

Specifies the number of descriptors in the LDT. pmEasy32 users generally have no need to change this since there is only a flat code and data segment. pmEasy16 users with many application segments or who allocate many blocks of memory will need to increase this. To see if this value can be reduced, consult the LDT display in the debugger at various points during program execution and note the highest descriptor allocated.

Diagnostics:

CHECKSTACK (pme.inc)

Set to 1 to cause pmEasy to check for stack overflow after the load. The load uses a great deal of stack compared to everything else done in pmEasy, so this is the pertinent time to do the check.. If the stack overflows or underflows, this is clearly indicated in an error message. Otherwise, the percentage of stack usage is displayed. If the usage is low for the driver(s) you use, you could reduce the stack size, PMSTACK_SZ.

coproPresent (pme.asm)

pmEasy16 only. This flag merely controls output of a warning message emitted by the NE loader indicating that coprocessor fixups were encountered in the application being loaded. The NE file format has a special fixup type intended to allow the loader to patch over the coprocessor opcodes with calls into the Microsoft Windows coprocessor emulator. This is not supported by pmEasy, so a warning is generated if these are encountered. If there is a coprocessor on the target, set this flag to 1 to disable the warning. Note that use of the C library might bring in routines that are unused (because they are in the same module as a routine that is used) that do

floating point operations and cause this warning even though your code does not. In this case, ignore the warning and disable it in your release, by setting this flag to 1.

debugpme (macro in pme.mak)

Set to 1 to allow debugging the initial, real mode portion of pmEasy in CodeView (supplied with Visual C++ and MASM). The bulk of pmEasy, including the loader, runs in protected mode and cannot be debugged with CodeView. However, there is a bit more real mode code now in pmEasy v2 so this might be helpful if you change it or want to add to it.

DIAGNOSTIC_MSGS (see the Disk Reader settings below)

Disk Reader: (see also Loader and Loading settings, below)

The settings in PCCONF.H control drive lettering, mostly. These should be left alone unless there is good reason to change them.

DIAGNOSTIC_MSGS (LOAD\DISKRD\disk.c)

Set to 1 to enable display of diagnostic messages in the disk reader, if you are having load problems that appear to be related to reading the disk.

DISKRD (pme.mak)

Set to 1 to enable the standard disk reader for loading from target.

DISKRD21 (pme.mak)

Set to 1 to enable the special disk reader that uses DOS int 21h services. When pmEasy is built with this option, it will only work on a system running DOS. This is the best choice for building a version of pmEasy to bind to the application. Prebuilt pmEasy EXEs using this option are supplied in the MISC\BIND directory to be bound to your application. The only reason to rebuild with this option is if you make changes to pmEasy.

ENABLE_CDROM_CODE (LOAD\DISKRD\disk.c)

Set to 1 to enable loading from CD-ROM. Set to 0 otherwise. Reading from CD-ROM requires a completely different file system than for reading DOS/FAT devices. Hence, DISKRD passes read requests to CDFS for drives that fall in the range of CD-ROM drives. By default, this range is set for 1 drive, E:. If your target does not have a CD-ROM drive and E: is actually another IDE drive type, you should set this to 0 so the read request will be handled by DISKRD and not passed to CDFS.

ENABLE_DISK_CODE (LOAD\DISKRD\disk.c)

This conditional is provided for those who are loading only from CD-ROM and want to eliminate the unneeded code for reading DOS/FAT disks. Set to 1 to enable the code; 0 to disable.

FIRST_ATAPI_CDROM_UNIT (LOAD\DISKRD\disk.c)

By default, this is set to FIRST_TRUE_IDE_ATA_UNIT + 2, meaning the master drive on the secondary IDE controller, or E:, assuming PCCONF.H is unchanged. If, for example, your first CD-ROM drive is the master on the primary controller, change it to +0.

RTFS (pme.mak)

Set to 1 to enable the RTFS disk reader. Leave this set to 0 unless this disk reader is obtained from us. It is no longer shipped, starting with pmEasy v2.

Disk Reader Drivers: (for DISKRD; macros in pme.mak)

cdfsdriver	(CD-ROM)
floppydriver	(Floppy disk)
idedriver	(IDE hard disk, LS-120, Iomega Zip®, CD-ROM)
msysdriver	(DiskOnChip)

Uncomment the macros for all drivers desired. When building for release, link only those needed. Note that if bootloading pmEasy, a subset must be chosen since enabling all drivers will make the image too large (must be < 64KB). This is also true of the debug version since it links the debug monitor.

FL_DMA_BUF_BEGIN and FL_DMA_BUF_SZ (pme.inc, pme.h)

These settings are for the Floppy driver. They control the size and location of the buffer used for DMA transfers. The DMA controller requires that the buffer be in real memory and not span a physical 64KB boundary. The default buffer is 9KB starting at the beginning of the buffer area. See the Buffer Area settings, above.

If FL_DMA_BUF_BEGIN is changed, DISKRD must be rebuilt.

DPMI Services:

DPMI100 and DPMI300 (pme.inc)

Setting these to 0 will eliminate the DPMI 100h and/or DPMI 300h functions, which will save code and data space. Most applications do not need these services and they can be disabled. An “Unimplemented DPMI Function…” error will be displayed if code in your application, such as a third-party library, attempts to use one of these. Also, eliminating DPMI 300h simplifies locating pmEasy to ROM/Flash (see *Running pmEasy from ROM* in the *Running pmEasy* chapter). See the comments where these are defined for more information about these routines.

realHeapSize (pme.inc)

Controls the size of the block of real memory used for heap by DPMI 100h.

Heap (Protected Mode):

GBAND_VAL (pme.inc)

Sets the value written to the first and last dword of each HCB to protect heap control blocks and written immediately above and below the protected mode stack to check for stack overflow and underflow.

HEAP_START (pme.inc)

Sets the beginning address of the heap. See also heapRegions (PME.ASM) for information about setting the end of the heap and any intermediate regions. Set this to the lowest linear address of the heap and use it in the first entry of the heapRegions table. As shipped, this is set to 10FFF0h (1M+64K-16 bytes) to avoid stomping on any DOS drivers, programs, etc that use the high memory area (HMA) so that

pmEasy will exit cleanly to DOS when done. (In real mode, if the A20 address line is set, one can use segment FFFFh to access the first 64K-16 bytes) of extended memory (the HMA). This is a trick long used by DOS to get a little more memory.) Conveniently, since the size of our HCB is 16 bytes, starting the heap at the byte just past the highest address accessible in real mode (10FFF0h) means that the first usable byte of the heap is at the even address 110000h. With the PE loader enhancements made in pmEasy v2, so that temporary memory is allocated from the top of the heap, the application image starts at 110000h, which is convenient when correlating addresses to the .map.

Note: If HEAP_START and heapRegions are set to start the heap in real memory ensure the heap does not overlap the buffer area used for the floppy driver DMA buffer or the saved _DATA image. (See the Buffer Area settings above for more information.) Set the ending HCB of the first region below the start of this area, so that this area is included in the gap for the PC ROM space.

heapRegions (pme.asm)

This table controls the memory used for pmEasy's heap (where DPMI memory allocations come from). See also HEAP_START. Each row of the table specifies a range of memory. The table is terminated by 0,0. Specifying 0 for the ending value of a range tells the heap initialization code in dpmiInit to scan for the end of RAM and set the end of the heap there. Example:

```
heapRegions   dd    HEAP_START, 00008fff0h
              dd    000110000h, 0
              dd    0,0
```

(HEAP_START is assumed to be < 8fff0h in this example.) This sets the heap to consist of 2 regions — one in real memory and one in extended memory starting at 1MB + 64KB. Note that the end of the second region is determined by a scan for the end of RAM. The scan simply probes the locations at the boundaries specified by SCAN_INCR (see below) and stops when the probe fails. The end of the heap is set where the last probe succeeded. Note that the ending address of a region is where the HCB will be written. Notice it is 10h less than 90000h for the first region above, since HCBs are 10h bytes in size. There is nothing magic about heapRegions — the gaps between regions are marked as allocated blocks. By ending the first region below 90000h in the example above, the area used by pmEasy for the floppy driver DMA buffer and to store _DATA for restart is avoided. It becomes part of the gap for the PC ROM space. The beginning of this area is controlled by

SCAN_INCR (pme.inc)

Used in automatic heap sizing. If you specify in the heapRegions table to scan for the end of RAM, this controls the increment in which the scan is done. The end of memory check simply writes a pattern and verifies that it wrote successfully. The last HCB is written where the last check succeeded. The smaller the value, the longer heap initialization will take. This feature could be used to verify memory integrity if set to 4, which would test every dword. However, this would take some time for a system with much memory. Typically, RAM sizing would be used when optional amounts of memory can be installed in a system so the scan boundary would typically be a multiple of the module size (e.g. 1MB, if RAM can be added in 1MB increments), or smaller. The default value works well.

Keyboard:

KEYBOARD (pme.inc)

If 1, pmEasy will wait for the Enter key to be pressed before starting the loaded application running. This is useful during development since it allows viewing the information displayed by the loader. When debugging, it gives time to start the debugger on the host. Set to 0 for release.

Loader and Loading: (see also Disk Reader settings, above)

appName (pme.asm)

This is the name of the application to load, including the drive it is on. The default is A:\APP.EXE. Change it to any drive and name you wish, but note that subdirectories are not supported, except by the DOS int 21h disk reader, DISKRD21. If you are not using DISKRD21, the path will be ignored, and pmEasy will attempt to load the application from the root of the drive specified. Note that drive lettering is not necessarily the same as DOS; see the section *Drive Lettering* in the chapter *Loading the Application*.

DEFAULT_DRVXT (pme.inc, pme.h)

Set to 1 to make pmEasy more lenient about the application name passed on the command line. This allows omitting the drive letter and extension, which will be supplied with defaults if necessary. (Must be used with PARSECMDLINE == 1.)

DefaultDrv, DefaultExt (utilpme.c)

These set the defaults for drive letter and file extension used by the DEFAULT_DRVXT option.

FNAME_SZ (pme.inc)

This sets the size of the buffer to hold the file name and path of the file to load, when specified on the command line. The default value should be large enough and probably can be reduced.

HEXLD_BUFSIZE (pme.inc)

Controls the size of the buffer used by the hex loader for processing the ASCII values into program bytes to be copied to the load destination in memory. Set to a multiple of 512, from 1536 to 64KB-512.

HEXLOAD (pme.mak)

Set to 1 to enable the HEX loader. Set all others to 0.

LOAD (pme.mak)

Set to 1 to enable the loader.

loader (macro in pme.mak)

Controls which of 3 loader .obj's is linked. These vary in the amount of messages displayed from all to none. Uncomment the version desired and comment out the rest. Note that there are 3 versions for each loader (PE, NE, HEX) so be sure to look in the right section. See also the VIDEO setting in PME.INC.

NELOAD (pme.mak (pmEasy16))

Set to 1 to enable the NE loader (default). Set all other loaders to 0.

PARSECMDLINE (pme.inc, pme.h)

Set to 1 so pmEasy will check to see if the name of the application to load was specified on the command line.

PELOAD (pme.mak (pmEasy32))

Set to 1 to enable the PE loader (default). Set all other loaders to 0.

quietLoad (pme.asm)

Leave set to 0. Disables loader output if set to 1, but the preferred method is to set the **loader** macro in PME.MAK (see above). quietLoad is an older method retained only in case a user finds it helpful to be able to quiet the loader dynamically rather than at link time.

READBUFSZ (pme.inc, pme.h)

Set to the desired size. This controls the size of the transfer buffer used by the DOS disk reader (DISKRD21). DOS reads into this buffer, in memory below 1MB, and pmEasy copies it to the destination address, usually in extended memory. 32K is the largest value that is effective. Experiment with different sizes and time the load to decide what size buffer to use, if you plan to use this disk reader in your production system.

UNLOAD (pme.inc)

Set to 1 to have pmEasy free the memory and descriptors used by the application after it is exited. This would be useful if pmEasy were modified to load another application after the first one exited. Otherwise, if just exiting to DOS or rebooting, this may be left set to 0.

Running pmEasy from ROM:

csiloc (macro in pme.mak)

Set this to 1 to locate pmEasy with CSi-Locate to run from ROM. Note that this is a technique that was implemented several years ago and has not been tested since, so if you wish to locate pmEasy to run from ROM, you should contact us to get b16rmsc.asm and any other modules and tips. As explained in the *Running pmEasy from ROM* section of the *Running pmEasy* chapter, this is not a standard option. See that section for more information.

Stack:

GBAND_VAL (see Heap settings above)

PMSTACK_SZ (pme.inc)

Specifies the size of the stack used by pmEasy in protected mode. This is the main stack used by pmEasy. The value has to be fairly large to accommodate use by the loader and disk reader. The size varies considerably depending upon which disk

device is loaded from. To tune this, set CHECKSTACK to 1 (pme.inc) and look at the percentage used displayed on the screen after the load has finished.

RSTACK_SZ (pme.inc)

Specifies the size of the stack used by pmEasy in real mode (the entry and exit code) and by DPMI 300h. The real mode code in pmEasy uses very little stack, so if DPMI 300h is not used, this can be small.

Target BIOS/DOS:

BIOS (pme.inc)

Set to 1 if the target has a PC BIOS. This adds code to turn off the floppy motor which may still be on from boot and to hide the BIOS cursor.

DOS (pme.inc, pme.h)

Set to 1 if the target is running DOS. This allows pmEasy to check for the presence of a DPMI server, and it allows pmEasy to return cleanly to DOS on exit.

Video Output:

VIDEO (pme.inc)

Set to 1 to enable video output. Generally this will be set to 0 in a production system, and PME.MAK will be set to link the quiet loader. 3 versions of the loader .obj are provided. The one linked is controlled by the *loader* macro in PME.MAK.

videoSeg (pme.asm)

Controls whether video output goes to color or monochrome video memory or to another RAM location. The value is the real mode segment address (B800h, B000h, or other), which is the linear address divided by 10h. Setting it to some other RAM location would be useful if there were no monitor on the target. This would permit viewing output in the debugger’s memory dump window.

DPMI API Reference

Routines are provided for memory management, local descriptor management, and interrupt management in conformance with the DOS Protected Mode Interface (DPMI) Specification. These routines are for use by your application and the interface to them will be described shortly. They are also used by pmEasy, itself. pmEasy supports a subset of the DPMI v1.0 specification developed by Intel and others and is available from Intel (order no. 240977-001).

The reasons for choosing to support the DPMI API are:

1. DPMI is a well-established standard.
2. Many third-party libraries use DPMI services, and offering a DPMI server enables pmEasy to support them.

The **memory management routines** allow allocating blocks of memory from pmEasy's heap. (See *Heap Structure and Initialization* in the *pmEasy Internals* chapter for details of the heap.)

The **local descriptor management routines** allow allocating, freeing, configuring, and cloning segment descriptors in the local descriptor table (LDT) . Application segments are mapped by the LDT. The global descriptor table (GDT) maps pmEasy and debugger segments. These routines are used by segmented applications to access allocated memory blocks. Generally, segmented applications should set the base address of a descriptor to the linear address of the block just allocated. The limit of the descriptor should be set to the block size minus 1 (base + limit is the last valid address in the segment).

32-bit flat mode applications do not need the LDT management routines since they load the segment registers with descriptors that map the entire memory space, so the linear address of the block returned by the memory allocation routine is used as the pointer to access it. This works because DS is 0-based. Although this saves the additional steps of allocating and initializing a new LDT descriptor for the block, it loses the protection that segmentation offers. This is one of the down-sides of flat mode development and is discussed elsewhere in this manual, where pmEasy16 and pmEasy32 are compared and where 16- and 32-bit protected mode are discussed.

The **interrupt management routines** allow hooking interrupt and exception handlers. Hooking a handler involves initializing a particular descriptor in the interrupt descriptor table (IDT) . The descriptor for a given interrupt number is set to have the address of the routine, and the type field is set to indicate whether it is a trap or interrupt gate and whether it is 16-bit or 32-bit. See *Writing Isrs* in the *Application Development* chapter for more discussion of 16-bit vs. 32-bit interrupt gates.

pmEasy16 provides a 16-bit DPMI server and pmEasy32 provides a 32-bit server. The two are mostly the same. The differences are primarily due to the need to pass and return 32-bit offsets for addresses (e.g. the address of the isr passed to pmiSetPMIntVect()). Another

difference is the need to be able to set large segment limits with the 32-bit server. The differences are pointed out in the descriptions of the routines below.

DPMI Assembly Interface

The DPMI specification lists many routines. Only a subset of them have been implemented in pmEasy. The routines are written in assembly and are contained in DPMI.ASM. To use them from assembly, particular registers must be loaded depending on the call. Then an int 31h instruction causes the routine to run. The values that must be loaded into the registers prior to the int 31h are specified for each call in the "Inputs" section in the comments ahead of each routine (and in the specification). AX is loaded with the number of the DPMI function to call. The values returned in the registers are documented in the "Outputs" section ahead of each routine. See the C API reference below for details of operation for each routine.

All registers that are not Inputs or Outputs of a DPMI function are preserved. Expect that the full 32 bits of the Input and Output registers could change. For example, AX is an input and output to all functions, but do not assume that the upper word of EAX is preserved. Similarly, if BL is an input or output, do not assume BH or the upper word of EBX is preserved. Save the register before making the call if its value must be preserved.

The following is an example of how to use the assembly interface. It shows how to initialize a descriptor to map an area of memory as a read/write data segment:

```
; allocate descriptor
mov     ax, 0h      ; allocLD
mov     cx, 1       ; allocate 1 descriptor
int     31h
mov     bx,ax       ; selector number. bx is input to all calls below
cmp     ax, 0       ; check for error code
jne     error

; set base address
mov     ax, 7h      ; setBaseAddr to
mov     cx, 0fh     ; cx:dx = 0f0000h
mov     dx, 0
int     31h
cmp     ax, 0       ; check for error code
jne     error

; set limit
mov     ax, 8h      ; setLimit to
mov     cx, 0       ; cx:dx = 0000ffff
mov     dx, 0ffffh
int     31h
cmp     ax, 0       ; check for error code
jne     error

; set access rights
mov     ax, 9h      ; setRights to
mov     cx, 92h     ; read/write data segment
```

```
            int     31h
            cmp     ax, 0           ; check for error code
            jne     error
            …
error:      …                       ; ax = error code
```

Calling from C

A C interface is also provided. See CPMI.H, which must be included in all C modules using these functions. The actual C DPMI functions are in CPMI16.ASM and CPMI32.ASM. Each function simply takes the values pushed onto the stack by the C calls, loads them into the proper registers for each DPMI routine, and then issues the int 31h. CPMI16.ASM or CPMI32.ASM must be assembled and linked with your application. Use the same assembler switches that are used in the pmEasy makefile. Note that pmEasy32 includes both modules because CPMI16.ASM is linked with pmEasy and CPMI32.ASM is for use by the application.

C API Summary

Memory Management Services

errCd **pmiAllocMem** (dword size, dword far *linAddrPtr, dword far *handlePtr);
errCd **pmiAllocMemTop** (dword size, dword far *linAddrPtr, dword far *handlePtr);
errCd **pmiBaseToHandle** (dword base, dword far *handlePtr);
errCd **pmiFreeMem** (dword handle);
errCd **pmiHeapStats** (HEAP_STATS far *statStructPtr);
errCd **pmiSizeMem** (dword handle, dword far *sizePtr);

LDT Descriptor Management Services

errCd **pmiAllocLD** (word num, word far *selPtr);
errCd **pmiAllocLDTop** (word far *selPtr);
errCd **pmiCreateAlias** (word sel, word far *aliasPtr);
errCd **pmiFreeLD** (word sel);
errCd **pmiGetBaseAddr** (word sel, dword far *linearBaseAddrPtr);
errCd **pmiGetDescr** (word sel, qword bufPtr);
errCd **pmiMapRealSeg** (word seg, word far *selPtr);
errCd **pmiSetBaseAddr** (word sel, dword linearBaseAddr);
errCd **pmiSetLimit** (word sel, dword limit);
errCd **pmiSetRights** (word sel, byte rights, byte xrights);

Interrupt Management Services

errCd **pmiGetPMIntVect** (byte intno, void (interrupt far * far *isrPtr)());
errCd **pmiSetExcepVect** (byte exno, void (far *esr)());
errCd **pmiGetIntDescr** (short sel, qword bufPtr);
errCd **pmiSetIntDescr** (short sel, qword bufPtr);
errCd **pmiSetPMIntVect** (byte intno, void (interrupt far *isr)());
errCd **pmiSimRealInt** (byte intno, struct IREGS far *regs);

Misc Services

errCd **pmeVersionNum** (word far *version);

where:

```
typedef  unsigned short int  word;
typedef  unsigned long  int  dword;
typedef  unsigned char  far  qword[8];  /* used for pmiGetDescr */
typedef  unsigned short int  errCd;
```

Although the details may vary, this gives the general idea. (See CPMI.H for details.) All routines use the return value for error codes. An error code of 0 means the call was successful. The error values are indicated in the descriptions of each call, in the following section. Many of the routines need to return one or more values. Since the return value is used for an error code, they do this through the parameters passed in. All parameters used for returning a value end in “Ptr” and are pointers to storage locations in the application. For example, a program calling pmiAllocMem() should allocate dwords to store the linear address and handle returned by the call. In making the call, the address of those storage locations is passed. For example:

```
...
dword   blockLinAddr;
dword   handle;
errCd   retval;
...
pmiAllocMem (0x100040, (dword far *)&blockLinAddr, (dword far *)&handle);
```

Other routines are similar.

DPMI / CPMI Reference

The following is a brief summary of each DPMI routine and its function number, for correlation with the Intel DPMI specification. Assembly users should refer to the comments above each routine in DPMI.ASM for information about the registers used for inputs and outputs for these routines. See the *DPMI Interface* section above for discussion about how to make DPMI calls from assembly. Since the C functions are just a wrapper around the assembly code, the functional descriptions here apply to calls from C and assembly. The routines are listed alphabetically.

The descriptions below are similar to the DPMI specification since our underlying DPMI routines follow the specification closely. However, some features are not relevant or were not implemented for other reasons. Do not assume that all features in the “Notes” section of each routine in the specification have been implemented unless explicitly mentioned here.

32-bit flat mode users should disregard the “far” and “interrupt” keywords in the function prototypes below. These are for 16-bit protected mode only. For flat mode, they are defined as null macros, since they are not supported by the flat mode compilers.

An inspection of DPMI.ASM vs. CPMI.H will reveal that several services are not made available in C. These are routines that are minimally implemented or stubbed off, in order to support some third-party library that makes use of these int 31h services, and we do not feel they are worthwhile to offer for general application use.

errCd **pmeVersionNum** (word far *version)

Func #: 04FFH (Non-Standard)

Purpose: Returns pmEasy's version number.

Descr: The version number is stored in **version*. It is of the form XX.X.X. For example, v2.0.0 is encoded 0200h. This service can be used by the application to determine which version of pmEasy is hosting it.

Errors: none

errCd **pmiAllocLD** (short num, short far *selPtr)

Func #: 0000H

Purpose: Allocates one or more descriptors in the local descriptor table (LDT).

Descr: If more than one descriptor is requested, descriptors are allocated contiguously and the selector of the first descriptor in the group (the one with the lowest value) is returned in **selPtr*. If there are not *num* contiguous descriptors, error 8011H is returned and none are allocated. Only the Present bit (7) and bit 4 are set in the descriptor(s) allocated. That is, byte 5 is set to 90h, which is the value for read-only data. All other initialization must be performed by other DPMI functions. Note that all bytes of unused descriptors in the LDT are zero; they are initially cleared and pmiFreeLD() clears them as they are freed.

Errors: 8011H descriptor unavailable

errCd **pmiAllocLDTop** (short far *selPtr)

Func #: 0080H (Non-Standard)

Purpose: Allocates the highest available descriptor in the local descriptor table (LDT).

Descr: This routine is intended to allocate a temporary descriptor that is soon to be freed. It was implemented for use by the pmEasy and smxDLM loaders so the temporary segments they needed for the load would not be sandwiched between those occupied by the loaded application or DLMs, leaving "holes" in the LDT upon completion of the load. In this way, the descriptors from multiple DLMs are immediately adjacent, reducing fragmentation of the LDT.

Generally, applications should use pmiAllocLD() instead, but this is offered for those who have similar use for it.

Errors: 8011H descriptor unavailable

errCd **pmiAllocMem** (dword size, dword far *linAddrPtr, dword far *handlePtr)

Func #: 0501H

Purpose: Allocates a block of memory from pmEasy's heap and returns both its linear address and a handle to it. The heap can be in real memory, extended memory, or both.

Descr: The linear address is the 32-bit address of the block, which is the same as its physical address since pmEasy does not currently support paging. This is passed to pmiSetBaseAddr() to set the starting address of the segment, except in flat mode, in which case this is the pointer to use to access the block, since DS maps the entire address space.

The handle is used to identify the block for other DPMI operations such as freeing and resizing and should not be used for any other purpose. The handle will be stored at **handlePtr*, if *handlePtr* is non-zero. This parameter is optional — if 0 is passed, the handle is not stored. The handle of any block can be obtained by passing its linear address to pmiBaseToHandle(), so it is not necessary to store the handle.

The block is guaranteed to be at least paragraph aligned. *size* is rounded up to the nearest paragraph. The minimum block size is 1 paragraph. An extra paragraph is allocated for every block to hold the control block for that memory block.

Note that this call does not allocate or initialize any descriptors in the LDT. It is the client's responsibility to allocate and initialize a descriptor using the LDT calls (see the list of calls in *C API Summary* preceding this section).

Errors:
8013H physical memory unavailable
8021H invalid value (size = 0)

errCd **pmiAllocMemTop** (dword size, dword far *linAddrPtr, dword far *handlePtr)

Func #: 0581H (Non-Standard)

Purpose: Allocates a block of memory from the top of pmEasy's heap.

Descr: Same as pmiAllocMem() but scans downward, starting at the highest address of the heap. It is intended for allocating temporary blocks of memory that are soon to be freed. It was implemented for the pmEasy and smxDLM loaders so that temporary blocks would not result in holes before the application and between DLMs in memory — a simple scheme to reduce unnecessary fragmentation.

Generally, applications should use pmiAllocMem() instead, but this is offered for those who have similar use for it.

Errors:
8013H physical memory unavailable
8021H invalid value (size = 0)

errCd **pmiBaseToHandle** (dword base, dword far *handlePtr)

Func #: 0599H (Non-Standard)

Purpose: Returns the handle of the heap block, given its base address.

Descr: The handle of the block is stored in **handlePtr* for use in subsequent DPMI operations that require a memory block handle, such as to free or resize the block. This function makes it unnecessary to store the handle at the time the block is allocated by pmiAllocMem().

Errors: 8023H invalid handle. After converting the base address to the handle, a check is made to ensure the handle is in range and that it seems to point to an HCB (a check of the fences is made).

errCd **pmiCreateAlias** (short sel, short far *aliasPtr)

Func #: 000AH

Purpose: Creates a new LDT descriptor with the same base and limit as the one specified by *sel*. The new descriptor is always a data descriptor with read/write access rights, regardless of the type of the original descriptor.

Descr: This is a one-time copy operation. If either the original or the alias is changed later, the other is not updated. The main purpose of this function is to allow read or write access to a read-only, code, or system segment.

Errors: 8011H descriptor unavailable
8022H invalid selector

errCd **pmiFreeLD** (short sel)

Func #: 0001H

Purpose: Frees a descriptor in the LDT, such as one previously allocated with pmiAllocLD().

Descr: This call always frees just one descriptor. If pmiAllocLD() was used to allocate several contiguous descriptors, each must be freed individually. Segment registers containing the selector being freed are cleared by this function. Attempting to use a cleared segment register before reloading it with a new selector will result in a gpf. The descriptor is completely cleared — most importantly, the Present bit is cleared.

Errors: 8022H invalid selector

errCd **pmiFreeMem** (dword handle)

Func #: 0502H

Purpose: Frees a memory block allocated with pmiAllocMem(). *handle* is the handle returned by pmiAllocMem() or obtained by pmiBaseToHandle().

Descr: The block is automatically merged with free blocks adjacent to it to form the largest possible free block.

Errors: 8023H invalid handle

errCd **pmiGetBaseAddr** (word sel, dword far *linearBaseAddrPtr)

Func #: 0006H

Purpose: Returns the base address of a segment.

Descr: The base address of the segment is stored in **linearBaseAddrPtr*.

Errors: 8022H invalid selector

errCd **pmiGetDescr** (short sel, qword bufPtr)

Func #: 000BH

Purpose: Copies the descriptor specified by *sel* into an 8-byte buffer.

Descr: This call is useful for displaying an LDT descriptor or inspecting fields or bits of it. Note that pmEasy defines a qword as the far address of an 8-byte character array. Define the buffer as “qword myBuf;” and pass “myBuf”.

Errors: 8022H invalid selector

errCd **pmiGetIntDescr** (short sel, qword bufPtr)

Func #: 0297H (Non-Standard)

Purpose: Copies the interrupt descriptor specified by *sel* into an 8-byte buffer.

Descr: Normally pmiGetPMIntVect() should be called to get an interrupt vector. This function is provided for cases where it is necessary to save and restore the entire 8 bytes of the descriptor. This can happen if assembly language were used in flat mode, for example, to directly access an IDT descriptor and set the vector to a far 32-bit address, such that the selector was not the flat code selector. Similarly, in a 16-bit application, a 32-bit offset could be set. A typical application will not directly access the IDT and should not need this special function.

Note that pmEasy defines a qword as the far address of an 8-byte character array. Define the buffer as “qword myBuf;” and pass “myBuf”.

Errors: none

errCd **pmiGetPMIntVect** (byte intno, void (interrupt far * far *isrPtr)())

Func #: 0204H

Purpose: Returns the address of the interrupt handler hooked at level *intno* in the interrupt descriptor table (IDT). For segmented protected mode, the address is in selector:offset form. For flat mode, it is a 32-bit offset. The segment is assumed to be the flat code segment.

In some cases, the whole interrupt descriptor (8 bytes) should be saved and restored. See pmiGetIntDescr() for more discussion.

Descr: This is useful for saving a vector to be restored later.

Errors: none

errCd **pmiHeapStats** (HEAP_STATS far *statStructPtr)

Func #: 0598H (Non-Standard)

Purpose: Returns several statistics of pmEasy heap usage.

Descr: Sets the fields of the structure pointed to by *statStructPtr* to indicate:

1. Total bytes of heap available
2. Number of free blocks
3. Size of the largest free block

This routine was coded very efficiently and should run quickly even if there are many blocks in the heap.

Returns 0 on success or non-zero if it failed. It could fail if the heap changed during its scan, and it could not continue following the chain of blocks because a control block got overwritten. Call this function until it returns 0. It is best to wait until a point in app when no heap activity, if possible.

Errors: non-zero if it fails

errCd **pmiMapRealSeg** (word seg, word far *selPtr)

Func #: 0002H

Purpose: Allocates and initializes a segment descriptor to map a real memory segment.

Descr: Creates a read/write descriptor with base == (seg * 16) and limit FFFFh, to map a 64KB block of memory below 1MB. This is a convenient way to create and

initialize a descriptor in one operation to access some area of real memory, such as video memory. For example, passing 0xB800 for seg will allocate a descriptor that maps the 64KB block starting at address B8000. The selector of the created descriptor is returned in **selPtr*.

Errors: 8011H descriptor unavailable

errCd **pmiSetBaseAddr** (short sel, dword linearBaseAddr)

Func #: 0007H

Purpose: Sets the 32-bit base address field of the LDT descriptor specified by *sel*.

Descr: Use this, for example, to initialize a new descriptor to the linear address of the block (not the handle) returned by pmiAllocMem(). This is the lowest memory address accessible through the descriptor.

Any segment register containing *sel* is reloaded with *sel* to cause the updated descriptor to be loaded into the hidden part of the segment register.

Errors: 8022H invalid selector

errCd **pmiSetExcepVect** (byte exno, void (far *esr)())

Func #: 0203H

Purpose: Sets the address of an exception handler, *esr*, in the interrupt descriptor table at level *exno*. Exceptions are faults issued by the processor, such as the general protection fault or division by zero.

Descr: Performs basically the same service as pmiSetPMIntVect(), except that *exno* is limited to the values 00-1Fh and the descriptor type is set to a **trap gate** rather than an interrupt gate. A 16-bit gate is created for pmEasy16 applications; 32-bit gate for pmEasy32 applications.

Segmented applications must pass the address of *esr* in selector:offset form. Flat mode applications pass a 32-bit offset. For flat mode, CPMI.ASM fills in the code selector since it is known a priori that there is only one code selector (0Ch).

Errors: 8021H invalid value (*exno* not in 00-1Fh range)
8022H invalid selector for *esr*

errCd **pmiSetIntDescr** (short sel, qword bufPtr)

Func #: 0298H (Non-Standard)

Purpose: Copies the 8-byte buffer to the interrupt descriptor specified by *sel*.

Descr: This function is used to set all 8 bytes of an interrupt descriptor. This function is not normally needed. Typical applications should instead call pmiSetPMIntVect(). See pmiGetIntDescr() for more discussion.

Errors: none

errCd **pmiSetLimit** (short sel, dword limit)

Func #: 0008H

Purpose: Sets the limit field in the LDT descriptor specified by *sel*.

Descr: The limit is the last address in the segment that can be accessed through the descriptor. Set it to the block size minus 1. For example, if a 64KB block is allocated with pmiAllocMem(), set the limit of the descriptor to 64KB-1 == FFFFh. The maximum limit for pmEasy16 is 64KB-1.

pmEasy32 applications can set limits up to 4GB-1, but for those 1MB and larger, the Granularity bit is set by this call, indicating page (4KB) granularity. This is necessary since the limit field has only 20 bits, rather than the full 32. The full 32 bits must be specified in the *limit* parameter and this routine automatically truncates the lower 12 and sets the Granularity bit. The specification requires that the low 12 bits be set. That is, *limit* must end in FFFh. (This is probably just a reminder that a limit such as 100000h is actually 100FFFh.)

Keep in mind that with limits of 1MB or larger, protection becomes coarse. If you allocated a block of 100700h bytes, for example, the tightest limit that could be set would be 100FFFh. The 900h bytes past the end of the block (100700h to 100FFFh) could be addressed by the selector. Those bytes might be in some other segment's space. For the best protection in segmented 32-bit applications (not yet supported by pmEasy32), it is best to allocate large blocks of memory in 4KB multiples, in this example, 101000h instead of 100700h, to ensure that the entire memory accessible by a descriptor is contained in one segment.

Any segment register containing *sel* is reloaded with *sel* to cause the updated descriptor to be loaded into the hidden extension of the segment register.

Errors:

8021H	invalid value. The upper word of *limit* must be 0 for pmEasy16. For pmEasy32, *limit* must have the lower 12 bits set (i.e. it must end in FFFh).
8022H	invalid selector

errCd **pmiSetPMIntVect** (byte intno, void (interrupt far *isr)())

Func #: 0205H

Purpose: Sets the address of an interrupt handler, *isr*, in the interrupt descriptor table at level *intno*.

Descr: All 256 entries in the interrupt descriptor table can be set with this call, even the trap entries (00-1Fh, see pmiSetPMExcepVect()). The descriptor type is set to an **interrupt gate** (even when setting the trap descriptors 00-1Fh). A 16-bit gate is created for pmEasy16 applications; 32-bit gate for pmEasy32 applications.

Segmented applications must pass the address of *isr* in selector:offset form. Flat mode applications pass a 32-bit offset. For flat mode, CPMI.ASM fills in the proper selector since it is known a priori that there is only one code selector (0Ch).

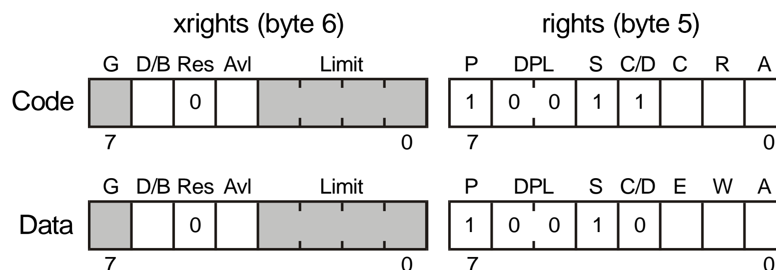
Errors: 8022H invalid selector

errCd **pmiSetRights** (short sel, byte rights, byte xrights)

Func #: 0009H

Purpose: Sets the access rights and type fields in the LDT descriptor specified by *sel*. pmEasy16 applications pass 0 for *xrights* (extended rights).

Descr: The *rights* parameter is the value for byte 5 in the descriptor, and *xrights* is byte 6. Shown in the order they appear in Intel's descriptor diagrams:



The **gray-shaded** bits are ignored in the value passed in. These are the Granularity and Limit, which are set by pmiSetLimit(). Regardless of the value passed, these bits of the descriptor are left untouched.

The bits with values shown in the diagram must be set as indicated or this function will abort with error 8021H.

Common Values:

rights:	9Bh	readable code
	92h	read/write data
	90h	read-only data
xrights:	0h	use16
	40h	use32

Definition of Bits (bold indicates preferred settings):

G: Granularity bit. Set only with pmiSetLimit().

D/B: Default/Big bit. For a code segment, this sets the default operand and address size used by instructions. For a data segment used as a stack or containing one, it controls whether ESP or SP is used and the stack upper bound. For pmEasy16, use 0; pmEasy32 creates the flat code and data segments with this bit set to 1.
0 = 16-bit code (use16) / SP, FFFFh
1 = 32-bit code (use32) / ESP, FFFFFFFFh

Res: Reserved. **Must be 0**.

Avl: Available. Not used.

Limit: Upper nibble of Limit. Set only with pmiSetLimit().

P: Present. (Set by processor; cleared by pmiFreeLD().)
0 = Not present. Descriptor is available.
1 = Present. Descriptor is in use.

DPL: Descriptor Privilege Level. **Must be 0**. 0 is highest and is currently the only level supported by pmEasy. That is, the application and pmEasy run at the highest privilege level.

S: System/Application. **Must be 1**.
0 = System descriptor (such as gate or TSS descriptor)
1 = Application descriptor

C/D: Code/Data
0 = Data. See E and W bits, below.
1 = Code. See C and R bits, below.

A: Accessed. Set to 1 by the processor when the segment is first accessed. The Intel documentation gives a case why one might want to clear the accessed bit of segments, but generally, it does not matter which way this bit is set, since it will be set to 1 by the processor when accessed.
0 = Not accessed
1 = Accessed

Data Segments:

E: Expand Up/Down
0=Expand up
1=Expand down

W: Writable
0 = Read only
1 = Read/Write

Code Segments:

C: Conforming.
0 = Non-conforming
1 = Conforming

R: Readable
0 = Executable only
1 = Executable and readable

Please see the *Segment Descriptors* section of the *Memory Management* chapter of the *Intel Programmer's Reference Manual*, for detailed discussion of these bits.

PME.INC defines several constants for common values for the rights parameters. Several of the most common ones are listed under "Common Values", above. Note that some, such as igate16, cannot be used for this function, since they are used to create system descriptors. They have the S bit (bit 4 of *rights*) cleared. Note that the Accessed ("A") bit is set by the processor when a descriptor is referenced, so 92h becomes 93h, for example. Keep this in mind when debugging and you notice the rights byte for a descriptor is not exactly as you set it.

After modifying the descriptor, any segment register containing *sel* is reloaded with *sel* to cause the updated descriptor to be loaded into the hidden extension of the segment register.

Errors:
8021H invalid value. Rights parameter(s) are invalid.
8022H invalid selector

errCd **pmiSimRealInt** (byte intno, struct IREGS far *regs)

Func #: 0300H

Purpose: Run a real mode software interrupt with interrupts disabled.

Descr: This routine disables interrupts and switches to real mode in order to run a real mode isr, such as a DOS or BIOS service. This is intended for use during initialization, not during normal system operation, in order to run hardware initialization code for which no source code is available. For example, this routine was initially implemented in order to run the BIOS int 10h video initialization routine, which is provided in ROM on the video controller card. Initialization differs for each card so it is a big benefit to be able to save writing this code. This is a general routine that can be used to run any real mode software interrupt routine.

Important: The _SS and _SP fields of the IREGS structure should be set to 0 or a valid real mode segment:offset address. If non-zero, the DPMI server will switch the stack to the specified address. The cpuFlags field should also be cleared because it is loaded into the flags register. If these fields have invalid values, it can hang or crash the system. You should set these fields or memset() the whole structure to 0 before initializing the fields needed by the interrupt function you are calling.

Errors: none

errCd **pmiSizeMem** (dword handle, dword far *sizePtr)

Func #: 050AH

Purpose: Returns the size of a memory block in pmEasy's heap.

Descr: The size of the block is stored in **sizePtr*. It is the size of the block useable by the client; it does not include the size of the HCB.

Errors: 8023H invalid handle

Error Reference

The following is an alphabetical listing of error messages that are displayed to the screen. The descriptions indicate causes and possible remedies.

Error messages from C routines generally appear in red on the right side of the screen, starting at the top. Long messages wrap across lines. Errors from assembly routines generally are shown in red (inverse video in versions prior to v2.23) on the left side of the screen along with Status messages, which are shown in normal, gray on black. Status messages are indicators that certain steps have been completed successfully. All screen display can be omitted by setting VIDEO to 1 in PME.INC. The color for error messages can be controlled by ERR_ATTR in pme.inc and pme.h. Consult a PC reference for details of the video attribute for setting foreground and background colors, or experiment.

pmEasy processor exception handlers display additional information, such as the instruction that caused the fault which is intended as a debugging aid for locating the problem. We plan to augment these to make them even more helpful in the future. Sections such as *Locating a GPF or Other Exception* show how to use the information provided to help pinpoint the cause of the problem. For errors marked "**Processor exception**," please refer to the Intel *386DX Programmer's Reference Manual*, Chapter 9, or a later x86 manual, for discussion of exceptions.

A20 NOT ENABLED

> pmEasy was unable to enable address line A20 and so is unable to access extended memory. Newer systems control A20 via port 92h. Older systems control it via the keyboard controller. pmEasy tries port 92h and then the keyboard controller. It reports this message if both attempts fail. This would indicate that the hardware has some other means to enable A20 or that something about it is non-standard. Contact Micro Digital.

ABORTED. DPMI SERVER ALREADY PRESENT.
PLEASE REMOVE IT.

> This error message indicates that pmEasy was run with another DPMI memory manager already present, such as Windows, EMM386, or 386MAX. Protected Mode memory managers conflict with pmEasy and must not be active when running pmEasy. Typically, they prevent pmEasy from entering protected mode and execution hangs.
>
> Note that the check that displays this error only catches protected mode memory managers that are DPMI servers. Other non-DPMI protected mode memory managers would not be detected. The symptom in that case would be that the last message displayed is "STARTING IN REAL MODE". See the chapter *Running pmEasy* for information about running pmEasy on Windows and DOS machines.

ADDRESS 0 IS 0

If reported when pmEasy starts, this means a program that ran before pmEasy had a bad pointer that overwrote address 0 (the x86 Divide by Zero fault vector in the IVT). pmEasy's RAM scan requires that address 0 and 4 be non-zero. As part of this check, each is changed to ADDR0_VAL if zero. If this is reported after the application returns to pmEasy, it means that the application pmEasy just ran has a bad pointer that overwrote address 0. You should debug your application to find where this occurs and fix it.

ALIGNMENT CHECK

Processor exception 17 (486+). *alignChk* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

APP LOAD FAILED

The application was not successfully loaded. pmEasy is aborted. Check the upper right area of the screen for other error messages. The problem may be minor, such as misspelling the file name or attempting to load from the wrong drive. The other error messages displayed along with this one will help pinpoint the problem.

APP NAME TOO LONG. INCREASE FNAME_SZ.

The buffer used to hold the file name passed on the command line is too small (due to a long path specified). Increase the FNAME_SZ constant in pme.inc.

App on which drive?

Loader error. The drive letter for the location of the application must be specified (e.g. A:APP.EXE). Check appName in pme.asm. Tip: if DEFAULT_DRVXT is 1, pmEasy will automatically prepend the default drive letter to the name. This saves keystrokes when entering the file name on the command line.

ARITHMETIC OVERFLOW

Processor exception 4. *ovrflw* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

CANT FREE FREE BLOCK

DPMI error. Reported by freeRealMem (101h) if the block is already free.

Checksum bad

Hex loader error. Reported if a record's checksum modulo 100h is not 0. The checksum is taken two hex digits at a time, which corresponds to one byte of data.

COPROCESSOR NOT AVAILABLE

Processor exception 7. *coproNavail* in pme.asm. If the EM bit of CR0 is set, the processor expects that floating point instructions will be handled in software and generates this interrupt as a hook into the emulator code. This message will only be seen if the emulator does not hook int 7 and no coprocessor is present. See *Locating a GPF or Other Exception* for tips on resolving this problem.

COPROCESSOR ERROR

Processor exception 16. *coproErr* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

DESCRIPTOR(S) UNAVAILABLE

DPMI error. *allocLD* (000h) could not allocate the number of descriptors requested from the LDT. If more than one is requested, there must be a *contiguous* block of the specified number or this error will be emitted, and no descriptors will be allocated. Increase *ldtSize* in pme.inc and rebuild pmEasy. Unless you need the descriptors to be contiguous, you can try to break the request into several requests for fewer descriptors.

DIVIDE ERROR

Processor exception 0. *divErr* in pme.asm. Divide by zero. See *Locating a GPF or Other Exception* for tips on resolving this problem.

DOS SECTOR CACHE NOT CLEARED …

pmEasy tries to clear the DOS sector cache on exit, in case the application made any changes to the disks. Clearing the cache causes DOS to reload sectors from disk rather than relying on the old data. Otherwise, DOS might cross-link files or otherwise corrupt the disk, and usually would display wrong data for directories or files. Additional information is printed after this message to indicate whether the reason is an old version of DOS (prior to v4) or a different vendor's DOS that works differently than expected.

DOUBLE FAULT

Processor exception 8. *dblFlt* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

EOF but no EOF record

Hex loader error. Reported if the physical end of file is reached before the EOF record. This would indicate that the .hex file has been truncated or corrupted.

Error opening disk

Loader error. The disk reader could not initialize the disk for access. Check that the drive letter specified in the application name is correct for the drive type. Note that pmEasy drive lettering is **not** the same as DOS drive lettering, unless DISKRD21 is used as the disk reader. Please consult the table in the comment next to the definition of *appName* in pme.asm, which shows the drive letters for supported device types. Also ensure that pme.mak is set to link the desired driver — that the macro is not commented out. Note that DISKRD21 supports any device supported by DOS, but it requires DOS and the DOS device driver to be present.

Error opening file

Loader error. The disk reader was unable to open the executable file to be loaded for some reason other than file not found, which is reported by a separate error message.

Error reading file

Loader error. The disk reader was unable to read data from the file. Possibly the file is corrupt. Try rebuilding the application. If loading from floppy, try formatting it first, or get a new one. Try loading from a different drive, especially a different device type. If the problem persists, contact Micro Digital.

Error seeking in file

Loader error. A seek operation failed in the disk reader. The executable may be damaged. Try rebuilding the application and see if the problem persists. If so, try loading from a different drive, especially a different device type. Contact Micro Digital if this problem persists.

File is not executable

Loader error. The file opened by the loader is not executable, is of the wrong format, or has been damaged. The file must be in the New Executable (NE) file format for pmEasy16 or the Portable Executable (PE) format for pmEasy32. These formats, and how to produce them , are described in Loading the Application.

File is not Hex

Hex loader error. The first character of a .hex file is expected to be a colon. If not, this error is reported.

File is not New Executable

pmEasy16 loader error. Indicates that the file is not of the New Executable (NE) file format. This format is an extension of the DOS .exe format and contains both the MZ and NE signatures. The letters “NE” can be seen in a text editor within the first screen-full. This format and how to produce it are described in the *Loading the Application* chapter.

File is not Portable Executable

pmEasy32 loader error. Indicates that the file is not of the Portable Executable (PE) file format. This format is an extension of the DOS .exe format and contains both the MZ and PE signatures. The letters “PE” can be seen in a text editor within the first screen-full. This format and how to produce it are described in the *Loading the Application* chapter.

File is truncated

Loader error. Reported by the loader on attempt to seek past the end of the file. The loader seeks to locations indicated in the header, so if this error occurs, it means that the file has been truncated or that the header has been corrupted.

File not found

Loader error. Reported if the loader could not find the file on disk. The name, path, or drive letter is bad. Note that DISKRD does not support subdirectories so the .exe must be in the root.

File to load is not specified

Loader error. Reported if no filename is passed to load().

Floating Point fixups encountered. Have coprocessor?

pmEasy16 loader error. The Microsoft and Borland floating point emulation libraries rely on WIN87EM.DLL. The compiler generates the coprocessor opcodes in the code and the linker creates fixup records (OSFIXUP) that point to all of these locations. If a coprocessor is present, the loader leaves the code as, is. If no coprocessor is present, the loader is supposed to patch over the coprocessor opcodes in the .exe with calls to routines in this emulation DLL. This does not work for pmEasy, since we cannot supply this Windows DLL. The user must have a coprocessor or use a third-party coprocessor emulator library, such as GoFast™.

Set *coproPresent* (PME.ASM) to 1 to disable this warning, if the target has a coprocessor or emulator. For some reason, we have found that although the smx Protosystem does not use math functions, a few from the Borland library get linked and these have OSFIXUPs. As a result, this warning message will always be generated for Borland 16-bit applications even if they don't do any math operations. (Unused routines will come in from a library if they are in the same module as a routine that is used.) Borland users should disable this warning.

FLOATING-POINT ERROR

Processor exception 16. *fpErr* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

GENERAL PROTECTION FAULT

Processor exception 13. *gpFlt* in pme.asm. This is the most common processor fault. It usually means that a memory reference exceeded the limit of a segment. It is also caused by loading an invalid selector into a segment register. The Intel x86 documentation lists the many other possible causes. See *Locating a GPF or Other Exception* for tips on resolving this problem.

HEAP INIT FAILED AT REGION [xxxxxxxx TO xxxxxxxx] CODE=xx

DPMI error. Reported by *dpmiInit* (dpmi.asm) if there is some error initializing the heap. Check the addresses specified in the *heapRegions* table (pme.asm). See the chapter *Return Code Reference* for a list of codes:

Image expects fixed base

pmEasy32 loader error. A field in the PE header of the .exe being loaded is set that tells the loader to only load the module at the preferred base address. This field is controlled by the /FIXED (Microsoft) or -B (Borland) linker option. Using these switches also omits the .reloc table, which prevents performing fixups. Fixups are unnecessary because the linker has already performed the fixups using the preferred base. The reason for using these switches is to reduce load time and the size of the .exe.

It is ok to use these switches, provided (1) the preferred base equals the actual image base and (2) pmEasy is not being bound to the application. This is explained in *Description of the PE File Format* in the *File Formats* appendix of this manual. If you get this error, you should set HEAP_START (pme.inc) so that the necessary condition is met (preferred base == image base) or do not use these switches.

For Microsoft, /FIXED is the default, so it is necessary to specify /FIXED:NO on the link line. In Visual Studio, this has to be manually added to the project settings. Go to Settings and select the Link tab. In the Project Options box at the bottom of the dialog, manually enter /FIXED:NO

INSUFFICIENT REAL MEMORY

DPMI error. Reported by *allocRealMem* (100h) if there is not enough memory for the allocation. Increase *realHeapSize* in pme.inc.

INVALID HANDLE

DPMI error. An invalid memory block handle was passed to a DPMI memory reallocation or free routine. Note that the handle of the block is returned by *allocMem* and is distinct from the pointer to the memory block. Ensure that you are not passing the pointer by mistake.

INVALID OPCODE

Processor exception 6. *invOpcode* in pme.asm. This error typically occurs if there is a bad jump, call, or return in the code that causes CS:IP to skip to an invalid address in memory, such as where there is no code, or between instruction boundaries. It can also occur if code gets corrupted. See *Locating a GPF or Other Exception* for tips on resolving this problem.

INVALID SELECTOR

DPMI error. The selector passed to the indicated DPMI function is out of range or the Present bit is not set in the descriptor. This bit is set by *allocLD* and cleared by *freeLD*.

INVALID TSS

Processor exception 10. *invTss* in pme.asm. *See Locating a GPF or Other Exception* for tips on resolving this problem.

INVALID VALUE

DPMI error. An invalid value was passed to the DPMI function indicated. See the description for the function in this manual and dpmi.asm, if necessary, to see what went wrong. BX and CX are displayed as an aid since they are most likely to contain the invalid value passed to the function. Most DPMI functions use these registers for input values.

Missing relocation table

pmEasy16 loader error. This error should not occur. It is a check done for confidence and indicates a damaged executable or a faulty link. The segment table entry for the segment being loaded indicates there should be a relocation table for the segment, but none is present. The relocation table for each segment immediately follows the segment. Try rebuilding the application and loading from another drive (in case it is a file read error). If these do not fix the problem, contact Micro Digital.

MSDOS CALL NOT SUPPORTED INT 21H FUNC: xx SUBFUNC: xx

This message appears if int 21h occurs. DOS calls are not supported by pmEasy. It is likely that your code has a C library call that makes a DOS call. We have provided a hook to allow implementing functions to emulate needed DOS int 21h services. See the module dosem.c. smx users should purchase unDOS if DOS emulation is needed. unDOS may be offered in the future for non-smx systems. If not, the smx version could be adapted by the user.

If DOS is present on the target, an alternative to writing emulation routines may be to use pmiSimRealInt (DPMI 300h) to actually run DOS int 21h services. This is done with interrupts disabled, so it should be used only during application initialization. See the documentation of this routine in the DPMI section of this manual.

Need more descriptors

pmEasy16 loader error. Reported if the loader cannot allocate enough LDT descriptors for the application. Increase *ldtEntries* in pme.inc.

Non-zero ADDITIVE fixup

pmEasy16 loader warning. If this occurs, please contact Micro Digital, so we can learn the purpose of this fixup type. The loader handles this as the documentation on the NE format directs, so your program may run fine. However, the documentation is very brief and we suspect it fails to mention a step. If there is a problem, please tell us so we can fix it.

Nowhere to store load information

Loader error. Reported if the *infoblk* parameter to load() is NULL. This is a pointer to a structure where the loader will store basic information about the application loaded, such as its entry point and information used to be able to unload the application from memory.

OUT OF BOUNDS

Processor exception 5. *bndsChk* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

Out of memory

Loader error. Reported if there is not enough memory to load the application, including temporary memory. Make the heap larger by changing HEAP_START (pme.inc) and the heapRegions table (pme.asm).

PAGE FAULT

Processor exception 14. *pgFlt* in pme.asm. See *Locating a GPF or Other Exception* for tips on resolving this problem.

PHYSICAL MEMORY UNAVAILABLE

DPMI error. *allocMem* or other DPMI memory allocation routine failed due to lack of memory. Make the heap larger, if possible, by changing HEAP_START (pme.inc) and the heapRegions table (pme.asm). The heap can span from real

memory to extended memory. See the discussion of heapRegions in the *Configuration Reference* chapter.

Problem with .exe

pmEasy32 loader error. Bit 2 of the PE header's characteristics field indicates (when set) that the .exe is ok to run. If not set, there were errors detected at link time or the image is being incrementally linked. This bit tells the loader not to load the .exe. Rebuild the .exe and see if that fixes the problem.

Record skipped? record/line #

Hex loader error. It appears that a record was skipped due to a damaged .hex file. After processing a record, the loader scans the buffer for the next colon. It should be closer than or at MAX_SKIP_DIST, or else it is likely that a colon was lost. Linker/locators typically put a CRLF sequence at the end of the record, so there should only be a maximum of 2 bytes to skip. If your linker/locator separates records by more than 2 bytes, increase this setting and rebuild the loader and pmEasy.

SAVE DATA FAILED, NOT ENOUGH SPACE

If RESTART is enabled, the debug version of pmEasy saves the _DATA segment to an area of RAM starting at DATA_SAV_BUF_BEGIN. A check is made to see if the data will fit within the upper boundary of the data area. If not, this error is reported. It is necessary for pmEasy to save the _DATA segment because some pmEasy variables change during pmEasy execution, and these must be returned to their original values in order to restart. This and related constants are in pme.inc and pme.h.

SEGMENT NOT PRESENT

Processor exception 11. *segNpres* in pme.asm. *See Locating a GPF or Other Exception* for tips on resolving this problem.

STACK EXCEPTION

Processor exception 12. *stkFlt* in pme.asm. This occurs on attempt to move the stack pointer outside the bounds of the segment whose selector is in the SS register. This could be due to too many pushes or pops or by manipulation of SP with add or sub. See *Locating a GPF or Other Exception* for tips on resolving this problem.

STACK OVERFLOW / UNDERFLOW

After the application load completes, if CHECKSTACK is 1, testing is done to ensure there has been no stack overflow or underflow. This condition is checked after the load, since the loader and disk reader are the main users of the stack. If stack overflow occurs, increase PMSTACK_SZ in pme.inc. Stack underflow would indicate some error in the handling of the stack pointer. This would most likely be caused by adding new code to pmEasy that has errors.

UNHANDLED PIC INTERRUPT IRQ xx

This message indicates that a hardware interrupt (irq0 to irq15) occurred and no handler has been set for it in the interrupt descriptor table (IDT). This is displayed by the generic handlers which are hooked to all 16 of the PIC interrupts in the IDT

during initialization. The irq number is indicated in the message. This is meant as a helpful diagnostic during development since it points out potential problems, such as: forgetting to hook an isr or an unexpected interrupt occurring.

Unhandled fixup type

pmEasy32 loader error. An unexpected fixup type has been encountered by the loader. Contact Micro Digital.

Unhandled relocation-address type

pmEasy16 loader error. **Byte 1** of the relocation record indicates an address type that is not handled by this loader. This byte specifies the type of address, such as a 16-bit selector, 16-bit offset, or a full 32-bit far pointer. Contact Micro Digital.

Unhandled relocation type

pmEasy16 loader error. **Byte 2** of the relocation record indicates a fixup type that is not handled by this loader. This byte specifies whether it is an internal reference, a reference to a DLL routine, or an "OSFIXUP". If the type is 7 or 3, this is called an "OSFIXUP," which is coprocessor fixup. See the "Floating Point fixups encountered" error above.)

Types 1, 2, 5, and 6 are for DLL references. Calls to DLLs are not currently supported. If one of these is reported, you may be using a function that expects the presence of a Windows DLL. If you cannot locate the code causing this reference, or if you cannot get rid of this error, contact Micro Digital.

UNHANDLED SOFTWARE INTERRUPT

This message indicates that an interrupt occurred and no handler has been set for it in the interrupt descriptor table (IDT). This is displayed by the generic handler which is used to initialize most of the IDT (all but the PIC interrupts).

UNIMPLEMENTED DPMI FUNCTION

DPMI error. An attempt was made to use a DPMI function which has not been implemented. Check dpmi.asm for a list of available functions. The contents of AX before the int 31h instruction gives the function number of the DPMI function to run. Ensure that AX has the intended value.

Unknown record type on record/line #

Hex loader error. An unexpected record type has been encountered or the .hex file has been corrupted.

Return Code Reference

dpmiInit Return Codes

dpmiInit primarily performs heap initialization. If unsuccessful, the heap region that was being initialized and the error code are displayed, if video output is enabled. The error code is also returned in the AX register. Summary of codes:

0	No error.
1	Failed writing hcb1 (start of region).
2	Failed writing hcb2 (end of region).
3	Failed writing prev_hcb.flnk. This is the link from the endof the previous region to the start of the next.
4	First check of RAM scan failed. This would suggest there is no RAM, even at the beginning of the heap. Check HEAP_START (pme.inc). Could be damaged memory. Try starting the heap a little higher or lower if HEAP_START looks valid.
5	Region is smaller than minimum block size.
6	Beginning address of region is above the ending address (i.e. the addresses are reversed)
7	Addresses must be on a paragraph (16-byte) boundary (last digit 0h).

Loader Return Codes

All loader errors are displayed if video output is enabled. Below is a chart correlating error numbers returned by the loader in the AX register, with their respective error messages. Please consult the *Error Reference* chapter for discussion of these errors.

0	No error
1	Error opening disk
2	Error opening file
3	Error reading file
4	Error seeking in file
5	File is not executable
6	File is not New Executable
7	Unhandled relocation-address type
8	Unhandled relocation type
9	Missing relocation table
10	App on which drive?
11	File is not Portable Executable
12	(obsolete)
13	Problem with .exe
14	Image expects fixed base (no .reloc)
15	Floating point fixups encountered. Have coprocessor?

16	File to load not specified
17	Nowhere to store load information
18	Need more descriptors
19	Out of memory
20	File name & path too long
21	File not found
22	register_file_user fail. Check NUM_USERS
23	Record skipped? record/line #:
24	Unknown record type on record/line #:
25	Checksum bad on record/line #:
26	EOF but no EOF record
27	File is truncated
28	File is not Hex.
255	Non-zero ADDITIVE fixup

Tips

1. 32-bit users can shorten application load time for Portable Executables under pmEasy32 by using the /BASE (Microsoft) or –B (Borland) linker option to locate the program to a preferred base and eliminate the need to perform fixups. Furthermore, /FIXED can be used for Microsoft to reduce .exe size by omitting the .reloc section. Borland's –B option omits .reloc too. See *Description of the PE Format* in this manual for details.

2. Borland C++ (32-bit) users should use v5 since older versions use a section alignment of 64KB, which wastes memory — each section starts on a 64KB boundary. BC v5 defaults to a 4KB alignment. All versions of Visual C++ have used a 4KB alignment.

3. The DISKRD21 disk reader can read from any device that DOS can read from since it uses DOS int 21h services. For applications larger than will fit on a floppy, it might be convenient to load from a larger removable storage device that may not be supported by DISKRD, which is intended for use on the final non-DOS target. Also, DISKRD21 is smaller (since all the work is done by DOS) and supports subdirectories, making it the disk reader of choice for binding to the application. See *Binding pmEasy to the Application* in the *Running pmEasy* chapter.

4. Assembly language: Except in rare cases, EXTRNs should be put inside segment definitions rather than at file scope to avoid the assembler making wrong assumptions about the address of the symbols.

Troubleshooting

As a first step, try running with **empty autoexec.bat and config.sys files** to ensure no DOS utilities are being loaded that could be causing the problem. In particular, memory managers and disk caches are problematic. If that fixes it, put back the original files and remove lines one at a time until the problem goes away, to isolate the cause.

1. Problem: STARTING IN REAL MODE is the last message displayed and execution hangs.

   Cause: Another memory manager is probably present that is conflicting with pmEasy as it tries to enter protected mode.

   Solution: Reboot with system files (autoexec.bat, config.sys) that do not invoke memory managers such as EMM386, 386MAX, or disk caches. Note that if a DPMI memory manager is detected, execution will abort with an error message. Consult the Error Reference if one occurs. pmEasy will run on a DOS machine or a Windows 9x/ME machine, if booted to DOS (not the MS-DOS command prompt in Windows). It will also run if bootloaded from disk by the BIOS.

2. Problem: HEAP INIT is the last message displayed and execution hangs. The loader has not yet run.

   Cause: Likely to be due to the RAM scan used to auto-size the heap. See *Auto-Sizing* in the *Heap Structure and Initialization* section of the *pmEasy Internals* chapter for explanation.

   Solution: Change the heapRegions table in pme.asm to explicitly set all region addresses. In particular, ensure no entry specifies 0 for the ending address. Initially try setting the end of the heap much lower than the amount of RAM present to see if this fixes the problem. For example specify an ending address of 0xFFFFF0 for a 16MB heap.

3. Problem: pmEasy won't load from E: or F:.

   Cause: PME\LOAD\DISKRD\disk.c routes reads to the CD-ROM reader for these drive letters, by default.

   Solution: Set ENABLE_CDROM_CODE to 0 in DISK.C, if you have ATA/ATAPI drives attached to the secondary controller that are not CD-ROM drives (e.g. hard disks, LS-120, Zip). Rebuild DISKRD and pmEasy.

4. Problem: pmEasy starts to load the program, but the computer reboots, or garbage is displayed on the screen and the load fails.

   Cause: Likely to be a stack overflow.

   Solution: Increase PMSTACK_SZ in pme.inc, rebuild pmEasy, and try again. Increase it a substantial amount. Then reduce it after a successful load, based on the percent used displayed on the screen.

5. Problem: 32-bit application not running.

   Cause: Possibly the entry point of the application is not specified. Starting with v2.1, pmEasy32 no longer assumes the entry point is at offset 0 and instead consults the PE header for the proper entry point. See the *Startup Code and Entry Point* section in the *Application Development* chapter for discussion of why this is.

   Solution: Specify the entry point after the END directive in the application startup code (e.g. startf.asm for smx systems).

6. Problem: pmEasy does not return to DOS.

   Cause: Might be that the floppy DMA buffer or _DATA image are overwriting memory used by DOS in the 90000h to A0000h area.

   Solution: Try lowering the address of BUFFER_AREA_BEGIN. Then rebuild the disk reader and pmEasy. See the Buffer Area settings in the *Configuration Settings Reference* section of the *Configuration Reference* chapter before changing these settings.

7. Problem: Code hooks irq vector but isr never runs.

   Cause: Maybe the wrong interrupt level was hooked. The traditional interrupt levels used for the master PIC (8h to 0Fh) cannot be used in protected mode because they conflict with the processor exceptions. In protected mode, it is necessary to reprogram the master PIC to generate higher interrupts since these values conflict with the processor exception vectors. pmEasy programs the master PIC to generate interrupts 40h to 47h and the slave PIC 48h to 4Fh. So, for example, the PC tick vector which is traditionally interrupt 8h is 40h in pmEasy.

   Solution: Hook the proper vector in the range 40h to 4Fh.

8. Problem: Descriptor access rights set to a value such as 92h but it is 93h, or 1 higher than expected.

   Cause: The 386 sets the Accessed ("A") bit of the descriptor when it is referenced. This is bit 0 of byte 5 of the descriptor.

   Solution: Not a problem.

9. Problem: Strange address displayed for gpf or other fault that looks like a real mode address (e.g. E900:B160).

   Cause: Can happen if a bad value is popped into the EFLAGS register due to a stack problem, such as unbalanced pushes and pops or corruption of an entry. If bit 17 happens to be set in the value that is wrongly loaded into EFLAGS, the processor is switched into virtual 8086 mode.

   Solution: Determine what is causing the stack corruption or other cause of EFLAGS being loaded with an incorrect value.

10. Problem: Build error: _TEXT segment overflow.

    Cause: The pmEasy code must fit into 64KB. If you have added code to pmEasy or you are building the debug version, which links the debug monitor, there is less space available for disk drivers.

    Solution: Eliminate unused disk drivers. Search for "idedriver" in the makefile to find the macros and comment out those that are unneeded.

11. Problem: pmEasy fails to build, and the unstripped log says "Out of memory" (in the directory with the .obj files).

    Cause: The PATH environment variable may be too long. From the command prompt, type "path" to see the current path.

    Solution: Shorten the PATH by deleting unused directories. Note that some program installs may create an autoexec.bat file that uses %path% and the result is that Windows duplicates the whole path. Comment out that line with "rem"and reboot. Check path again. If you cannot shorten the path, create a new one in the .bat file you run to create the MC16 command line environment that has just the paths you need.

Appendix A: File Formats

New Executable File Format (NE)

Description of the NE File Format

The New Executable (NE) file format is an extension of the DOS .exe format, created by Microsoft for Windows applications. It starts with the same header used for DOS EXEs and has the ".exe" extension. You may have seen the message "This program requires Microsoft Windows" if you accidentally tried to run an NE file from DOS. This message is printed by the stub program that is automatically linked to the .exe. Any real mode program can be bound to the .exe in this way. When pmEasy is bound to the application, this is how it is done.

If you look at any .exe with an editor, you will see that its first 2 bytes are the characters "MZ". If you look a few lines down, you may also see "NE". This signature indicates that the file is a new executable and marks the start of the Windows header. The Windows header indicates important information about the contents of the file. There are many fields, which specify where other parts of the file are located and how long some of the components are.

Because the NE format has to handle the complexity of Windows applications, such as resources and dynamic link libraries (DLLs), there is quite a bit of information packed into the file. The header has many fields, and there are quite a few tables in the file. Since your embedded application will not make use of such Windows features, many of these fields and tables are unneeded, and the pmEasy loader ignores them.

The parts of the NE referenced by the pmEasy loader are the Information Block (header), Segment Table, the code and data, and the relocation tables. Each segment of the application is kept separate in the file, unlike the DOS .exe format which clumps them into one block. Immediately following each segment is a relocation table, indicating the locations of all far addresses in that segment that need to be fixed up at load time.

Performing the fixups necessitated by the relocatable nature of the NE format is one of the most important jobs the loader does. An example is fixing a reference to a selector to have the proper value for the selector. In the .exe, segments are numbered 1,2,3,..., but when loaded, their corresponding selectors might be 14h,1Ch,24h,.... Wherever your code references segment 2 in a far address, for example, it must be fixed-up to have the proper value of 1Ch (or whatever selector was chosen by the loader for segment 2). Besides selector fixups, there are also 32-bit pointer fixups, where both a segment and offset need to be fixed-up.

Some linkers form chains of fixups, such that each fixup spot has the offset to the next fixup spot of the same kind. This saves many relocation table entries, which makes the file shorter and the load quicker.

Some of the features of the NE file format are not well-documented in the Microsoft documentation and other documents we've found. Most importantly, some of the fixup types that could be used are not explained. If an unusual fixup type or other load problem is encountered while loading the application, the pmEasy loader will display an error message. If this occurs contact us.

If you are curious to learn more about this format, or want to modify the loader, you can create a hex dump of an EXE, as discussed in the next section.

Examining NEs with EXEHDR and TDUMP

These are utilities for displaying information about an .exe, in a readable form. They are provided with the Microsoft and Borland C/C++ compilers, respectively. TDUMP was mentioned previously since it has the additional capability of producing a hex dump of the entire file. These utilities show the locations and sizes of the segments, as well as their attributes. They also show the fixups for each segment. We suggest you redirect the output to a file. For example:

```
exehdr -verbose app.exe > hdr     or

tdump app.exe > hdr
```

Portable Executable File Format (PE)

Description of the PE File Format

Like the new executable (NE) format, the Portable Executable (PE) format is an extension of the, DOS (MZ) .exe format. If you try to run one of these from DOS, its stub program will say "This program cannot be run in DOS mode" or something similar. As with the NE format, pmEasy can be bound to a PE file, and it is linked in place of this default stub program. See the subsection below for more discussion of this.

A PE file has a 4-byte signature: "PE/0/0" ("PE" followed by two null bytes). Most of the fields of the header and other structures are also double words.

The most significant differences between the PE and the NE formats are flat model vs. segmented model. Specifically, PE files use 32-bit offsets and there are no segments.

The PE header follows the MZ header, just as for the NE format. Technically, there is the header and the "optional header," but the two might as well be considered a single header, since they are contiguous and both are required. In fact, there is a lot more crucial information in the optional header than in the first header — hardly optional. (Probably a different meaning was intended for "optional.")

The file is broken into sections. These have names such as .text, .data, .reloc, and .debug. There are 8 or 9 standard sections and each PE file has some subset of them. (Some linkers use different names for some sections. Here we list the standard names.) In addition, new sections can be created in assembly language in the same way that segments are defined in segmented programs. If no user sections are defined, all code is put in .text and all data in .data, .bss, and .rdata. The .reloc section contains the fixup tables. Other sections are ignored by the pmEasy loader. A table immediately follows the PE header, with an entry

for each section, giving information such as its length, name, and location in the file. Each entry in the section table is only 40h bytes, so the section table easily fits within a 4KB page.

When loading, sections start on page boundaries. That is, the minimum section size is 4KB. Even a simple program may have 5 sections, which would mean that its image is 20KB, minimum. Small programs do not fare well with this format! The up-side is that very large programs are supported, and for them memory wastage is relatively insignificant. Versions of the Borland 32-bit compiler before 5.0 produced EXEs with a 64KB section alignment. Our hypothetical 5-section program would need a minimum of 320KB of memory! Borland v5 defaults to a 4KB section alignment like Microsoft.

Performing fixups is much simpler for PE files than for segmented files such as NE files. The same fixup is applied to every fixup site in the file. This is because there are no segments, and all addresses in the image are 32-bit offsets. Sections, unlike segments, have no significance in performing fixups. It is not important that a reference is made to another section, since all addresses are merely offsets relative to a common point in memory. A simple thing would have been to base them at 0. Instead, there is a field in the header which defines the *preferred base* and all offsets in the .exe are relative to it.

Fixups are performed by adding a delta to each offset in the image. The delta is the actual base minus the preferred base. This adjusts all offsets up or down a constant amount depending upon whether the image was located above or below the preferred base. If the image is loaded at the preferred base the loader skips the fixup step. In this case, the fixup table is not even loaded. A little load time can be saved by specifying the preferred base with the appropriate linker switch. This would be especially noticeable when loading from floppy or other slower media. Note that this technique cannot be used when pmEasy is bound to the application. See *Notes about Binding pmEasy to a PE*, below, for more discussion.

The linkers allow setting the preferred base only to a 64KB boundary. Since pmEasy always allocates memory for the image from the start of the heap, the address of the block is constant. By adjusting the start of the heap to 10h bytes below a 64KB boundary, you can ensure the image will start on the 64KB boundary. The 10h allows for the heap control block. The switch for Microsoft link is /BASE; for Borland tlink32 it is -B. Microsoft users should additionally use /FIXED:YES to omit the .reloc section, making the .exe smaller. The Borland –B option does this too.

If you are interested to learn more about the PE format, the Microsoft Developer Network CD has a thorough article, titled *The Portable Executable File Format from Top to Bottom* by Randy Kath. Search the entire CD for “Portable Executable” to find it and other references. Strangely, this article is quite in-depth, yet does not discuss the fixup table and how to perform fixups. All sections but .reloc are documented. For this, we relied on a separate specification of the format which we obtained from the Intel TIS department (Tool Interface Standards) in printed form (order no. 241597-001, 1-800-548-4725).

Examining PEs with DUMBPIN and TDUMP

DUMPBIN is a handy utility provided by Microsoft with 32-bit Visual C++ that shows various information about a given PE. The /headers option displays header information only. The /all option displays full information about the file, including the header and even hex dumps of the sections. Use the /? option to list other options that will yield more specific information about the PE file. Example usage:

```
dumpbin /headers app.exe > hdr
```

Borland's TDUMP utility provides similar output. Example usage:

```
tdump app.exe > hdr
```

Notes about Binding pmEasy to a PE

Although the stub program appears before the PE signature and section table in a PE file, for some reason, the Borland and Microsoft linkers (and probably all others) treat the stub program as part of the protected mode program image. All sections are shifted higher by the size of the stub program (rounded up to section alignment). Nothing actually gets loaded in that space, so it is wasted. This causes trouble for debugging, since the addresses in the debug symbolics will not match the usual address specified in the .CMD file passed to CSi-Locate. We wanted the application to be loaded the same way regardless of whether pmEasy was bound to it or not, and we wanted to avoid wasting memory needlessly, so we enhanced our PE loader to shift the sections back down in place of the stub. To do this, the loader adjusts several fields in the copy of the PE header in memory, the base addresses of each section in the section table, the fixup locations, and the delta applied for the fixups.

If you wish to avoid loading and processing the fixup table by forcing the preferred base to be equal to the base address where pmEasy loads it, you should not bind pmEasy to the application. The linker has already set all addresses assuming presence of the stub, so when the loader shifts the image down to occupy the space that would have been wasted by the pmEasy image, it must shift all addresses down by the size of the image, which will typically be 30K or so, not a multiple of 64KB (the required alignment of the preferred base setting). That is, it must perform fixups, even if the preferred base and image base are equal. (With clever setting of the heap base and preferred base and possibly forcing the pmEasy image to an even 64KB in size, it might be possible to avoid fixups, but we don't recommend this unless you have good motivation and time to experiment.)

The loader normally shows the image base and preferred base addresses in bright white when they match, but they always appear dimmed if there is a stub program linked, other than the tiny one that prints "This program must be run under Win32".

Appendix B: Boot Sector

This appendix explains a few details about the boot sector and boot sector program.

Boot Sector Layout

The boot sector is organized in 2 parts:

1. Boot Parameter Block (BPB) which contains drive parameters (number of heads, sectors per track, etc.)
2. Boot Program

The first 3 bytes of the boot sector are a JMP instruction to skip over the BPB to the start of the boot program. Adding 2 to the second byte gives the offset of the start of the boot program. The last 2 bytes of the boot sector are the usual DOS AA55h validity marker. The BPB is written by the format utility and is untouched by our BOOTWRIT utility — only the boot program is replaced.

Both the BPB and boot program vary for FAT32 disks vs. FAT12/FAT16 disks. Several fields in the BPB are wider for FAT32 and there are several new ones. The layout of a FAT32 disk is a little different since the root directory is not a special region, as in FAT12/FAT16 and is instead part of the data area of the disk. Because of these differences, a different boot program is needed for FAT32. Since there is still a significant amount that is common, we were able to create a single module, BOOTSECT.ASM, with conditional sections and generate both boot sector program images from it.

Restoring the Boot Sector (MBR)

If you accidentally run BOOTWRIT on the hard disk of your development system the following notes may be helpful to reverse the damage so it will again boot to Windows or whatever other operating system you are using. These are meant as tips and helpful information only. Micro Digital assumes no responsibility for damage you do to your disk while attempting to restore it with these or other methods.

WIN9X and DOS systems: Run "sys c:".
WINNT: We have not investigated.

It is likely that there are third-party utilities for restoring the boot sector. Failing that, try DISKEDIT in the Norton Utilities. Sometimes a backup copy of the boot sector is stored in a nearby sector. Check the hidden track (the one with the partition table). To see it, view Physical Sectors and specify 0 for cyl, head, and sector. Then page down through the sectors. The first is the partition table. If there is no saved copy, you could try copying the boot sector from another machine — **but don't overwrite the drive parameters (BPB)** at the start of the sector! The start of the boot program is easily obtained. The first 2 bytes of

the boot sector are a short jump to it. Add 2 to the second byte to get the offset. Example: EB 3C. Boot program starts at 3E. NT users can run a DOS/Win9x version of DISKEDIT from a DOS boot floppy. If the C: drive is formatted as NTFS, select physical disk 1. To do this, select File | Object, click the Physical radio button and then select the first hard disk. Good luck.

Appendix C: DiskOnChip®

The M-Systems DiskOnChip flash disk is an attractive device for use with pmEasy. DiskOnChip 2000 and Millenium are both supported. The driver is based on TrueFFS v5.1.0 released by M-Systems May 15, 2002. It does not support Millenium Plus or Mobile DiskOnChip, nor larger DiskOnChip 2000 (>192MB (low density) and > 384MB (high density).

1. DiskOnChip is a bootable device. pmEasy can bootload from it and load the application from it.
2. smxFile supports it, and since it is offered in large sizes, it can be used as the sole storage medium for the device, both for booting and application data storage. Since it is a solid state device, it is more rugged and consumes less power than a mechanical disk drive.
3. It is easy to update pmEasy or the application during development or in the field if a floppy drive is connected or can be temporarily connected. The system can be booted to a DOS floppy and DiskOnChip is accessible in DOS, like any disk, so PME.SYS and APP.EXE can be overwritten simply by copying the new files.
4. It is available in a range of sizes from a couple megabytes to a couple hundred megabytes and all are pin-compatible, meaning different sized units can be inserted at production time as an option.
5. It is commonly used on off-the-shelf PC-compatible embedded boards.
6. Board real estate required is minimal, compared with flash arrays.

This appendix covers a few topics related to use of DiskOnChip with pmEasy.

Procedure for Programming DiskOnChip

Note that for production, DiskOnChip can be gang-programmed.

1. Boot the target to a DOS floppy.
2. If the DiskOnChip already appears as drive C: and files can be copied to it, skip to step 4. Otherwise, format the drive with DFORMAT (available from the M-Systems web site):

   dformat /win:d000 /s:doc51.exb /first

   D000h is the location of the DiskOnChip window (segment), which varies depending on how the board is jumpered. Run DINFO to get the address if not known. Divide by 0x10 since it is specified in paragraphs on the command line. doc51.exb is the DOS driver. The name varies depending on its version number. /first is what makes it the boot device — it installs as C: rather than D:, so the PC boots to it. You may give it a volume

label by specifying the label switch on the command line to DFORMAT. For example, /label:diskonchip gives it the volume label “DISKONCHIP”.

3. Reboot to a DOS floppy. Ensure C: is the DiskOnChip. The reboot is necessary so that DOS will now recognize the DiskOnChip and assign a drive letter to it.
4. Run: bootwrit c:
   This is our utility, supplied in PME\BOOT.
5. Create PME.SYS with the DOS EXE2BIN utility, and pad it to 64KB with our PAD utility. See *Bootloading pmEasy from Disk with BIOS* in the chapter *Running pmEasy* for details.
6. Copy PME.SYS to the DiskOnChip.
7. Copy APP.EXE to the DiskOnChip.
8. Remove the floppy and reboot.

DiskOnChip Organization

DiskOnChip is organized into 2 parts: the file area and a hidden area. The hidden area contains the DOS device driver. During a DOS boot, this driver loads and allows DOS to access it as a normal disk device. This driver can be updated or restored with M-Systems utilities. Please refer to their documentation.

DiskOnChip Boot Sector

In order for the DOS device driver to load successfully, a few conditions must be met in the boot sector. This is because the driver scans a few places in the boot sector as a validity check and aborts if not satisfied. If the driver aborts, the DiskOnChip is invisible to DOS; no drive letter is assigned. Of course, M-Systems made it work for standard boot sectors such as those written by the DOS format utility. Since we overwrite the boot sector it was necessary for us to determine what the rules are. Here is what we found:

1. The JMP instruction at the start of the boot sector is checked to ensure that it is of the form E9 xx xx or EB xx 90.
2. The OEM ID field is also checked. We did a lot of experimentation to learn more about the check. The simple solution is to set all 8 bytes to alphanumeric characters (no spaces or other characters). To use another pattern, here are the rules: The 7th digit must be set to a value of 0x2F or greater. The period character is 0x2E so it is a problem. However, it can be used in the 7th digit if the 6th digit is set to an alphanumeric character. For example “MSDOS5.0" is valid, but "MSDOS .0" is not. Our BOOTWRIT utility lets you set the OEM ID field (search BOOTWRIT.C for "OEM" to find this).

Index

.abs file, 39
.cm file, 39
.debug section, 23
.def file, 12
.reloc section, 23, 89

/BASE, 97
/FIXED, 89, 97
/STUB switch, 12

_MM_, 30

16-bit applications, 27
16-bit vs 32-bit applications, 4

256-byte alignment (PE), 23

32-bit applications, 27

A20, 45, 48, 60
altering pmEasy, 55
APP16.CMD, 22, 38
APP32.CMD, 22, 38
application, 28
  entry point, 30, 100
    16-bit applications, 30
    32-bit applications, 30
  flat mode, 69
  segmented, 69
  startup code, 30
  unloading, 67
application development, 27
application name, 66
  precedence of options for determining, 13
appName, 9, 13
autoexec.bat, 10

-B, 89, 97
binding pmEasy
  to Portable Executable (PE), 106
  to the application, 11
binding pmEasy to the application
  how to, 12
BINDING_SUP, 12, 13
bindpme macro, 12
BIOS, 68
BIOS call, 83
BIOS extension, 16
BLDID, 46
BLDSD, 45
boot program, 107
boot sector, 107
  boot program, 107
  BPB, 107
  DiskOnChip, 110
  FAT32, 107
  layout, 107
boot sector program, 14, 107
bootable disk, directions to make, 11, 14
BOOTSECT.ASM, 15, 107
BOOTWRIT, 14, 15, 107, 110
Borland compiler, 4, 31
BPB, 107
BSEC.BIN, 15
BSEC32.BIN, 15
building pmEasy (directions), 4

C library, 30
  keeping out startup code, 30
CDFS, 21, 63
CD-ROM, 19
  driver, 21
  loading from, 63
CHECKSTACK, 22
code development, 27
CodeView, 57
command line, 67
  passing application name on, 10
config.sys, 10
configuration settings, 60
  cross reference, 59
configuring pmEasy, 59
coprocessor, 89
coprocessor emulator, 86
copying app to target, 13
CPMI.H, 71
CPMI16.ASM, 71
CPMI32.ASM, 71
CR0, 46
CSICFG, 38, 61
csidir macro, 39
CSi-Locate, 7, 22, 38
CSi-Mon, 37
CSIMON directory, 1, 37

Index

DBG directory, 1
debug symbolics, 7, 38
debug version, 27
debugger
  baud rate, 38
  COM port, 38
  restarting the application, 40, 62
debuggers, 7
debuggers supported, 37
debugging pmEasy, 57
  protected mode code, 57
  real mode code, 57
debugging the application, 37
DEBUGVER, 29, 47
default drive and extension, 10
DEFAULT_DRVXT, 10, 13
descriptor
  access rights and type, 80
  alias, 75
  base address, 78
  copying, 76
  copying interrupt descriptor, 76
  flags, 80
  freeing, 75
  set limit, 79
  temporary allocations, 73
device drivers, 19
directory structure, 1
disk driver stubs, 20
disk drivers, 20, 21
  CD-ROM, 21
disk readers, 20
  DISKRD, 12, 20, 21, 63, 88
  DISKRD21, 20, 67, 87, 97
  RTFS, 20
DiskOnChip, 19, 109
  boot sector, 110
  organization, 110
DiskOnChip, reboot if changed, 49
DISKRD, 12, 20, 21, 63, 88
DISKRD21, 20, 67, 87, 97
disks supported, 19
DLM's, 22, 23
DOS, 68
DOS call, 83
DOS command prompt, running from, 9
DOS emulation, 31, 91
DOS extenders, 4
DOS high memory area, 64
DOS sector cache, 49
dos21Em(), 31
DOSEM.C, 31
DPMI 300h, 16, 83, 91
DPMI API, 69
DPMI interface
  return code, 72
DPMI reference, 72
DPMI server, 68, 69
  assembly interface, 70
    example, 70
  C API summary, 71
  C interface, 71
  registers preserved, 70
DPMI server conflict, 10, 85
DPMI v1.0 specification, 69
DPMI.ASM, 70
dpmiInit return codes, 89, 95
drive lettering, 20, 87
driver
  floppy DMA buffer, 64
driver selection, 21
driver stubs, 20
drivers, disk, 20
DUMPBIN, 105

entry point, 22, 30
environment block, 11, 13
error reference, 85
exception handler, 42
  set vector, 78
exception handlers, 85
EXE2BIN, 14, 15
EXEHDR, 104
exiting pmEasy, 48
EXTRN, 97

far addresses in pmEasy, 55
far pointers, 54
FAT12/FAT16, 15
FAT32, 15, 107
file formats, 103
file organization, 1
fixup table, 89
  (PE), 23
fixups, 35, 55, 105
Flash
  loading from, 22
flat code and data segments, 22
flat mode, 27
  code and data segment initialization, 35
  segments, 34
floppy disk, 19

GDT, 69
  initialization, 45
GDTR, 45
GoFast, 23, 25
graphics mode, 48
guard band, 64

hcb, 51

HCB
- blnk, 51
- diagram, 51
- fence, 51
- flnk, 51
- in-use bit, 51

heap
- allocating, 74
- auto-sizing, 52
- diagrams, 52
- initialization, 51
- regions, 52
- scan for end, 52
- scan for end of RAM, 65
- separate memory regions, 52
- start, 64
- statistics, 77
- structure, 51

heap control block, 51
- diagram, 51

heap start and DOS, 48
HEAP_START, 22, 38, 52
heapRegions table, 52, 65
Hex loader, 25, 66
HMA, 64

IDE hard disk, 19
IDT, 34, 69
- creation, 46
- set descriptor, 79
- set exception handler, 78
- set interrupt handler, 80

IDTR, 45
include files
- order of, 29

installation, 1
int 21h, 31, 91
int 21h file i/o, 12, 20
int 31h, 70
INTERLNK, 14
internals, pmEasy, 51
interrupt descriptor
- setting, 79

interrupt descriptor table, 34
interrupt gate, 31
interrupt handler
- setting, 80

interrupt keyword, 32, 56
interrupt management, 69
interrupt vector
- getting, 77
- saving and restoring, 34

interrupt vector table, 34
isr
- Borland 32-bit, 32
- Microsoft 32-bit, 32
- shell, 32

isr doesn't run, 100
ISRs
- saving and restoring interrupt vectors, 34
- writing, 31

ISRs in pmEasy, 56
IVT, 34

JMP16, 46

keyboard, 66
- disabling, 9, 17

LD_INFO_BLK, 47
LDT, 46, 69
- descriptor
  - allocating, 73
- management, 69

LDTR, 46
LGDT, 45
LIDT, 46
linker/locator, 4, 25
load devices, 19
LOAD directory, 1
load time, shortening (PE), 97
loader, 66
- selecting version, 66
- temporary memory, 22

loader return codes, 95
loader screen display, 24
loaders, 22
loading from CD-ROM, 21
loading from other media, 20
loading the application, 19, 47
- from disk, 19
- from other sources, 21

located application, 26
locating GPF or other exception, 42
locator, 16
LS-120, 19

makefiles, 1
MASM, 7
MBR
- restoring, 107

media supported, 19
memory
- allocating, 74
- base address of block, 76
- block handle, 75
- block size, 83
- freeing block, 76
- temporary allocations, 74

memory management, 69
memory manager conflict, 10
memory model, 27, 30

Index

pmEasy, 54
Microsoft compiler, 4, 31
Microsoft Visual C++ (16-bit), 7
MINAAPP, 27
building, 27
directory, 1
running, 28
MINCAPP, 27
building, 27
directory, 1
running, 28
minimal applications, 27
building, 27
defines used in makefiles, 29
running, 28
MISC directory, 2
module inclusion order, 29
M-Systems DiskOnChip, 109

naked attribute, 32
NE loader, 22, 23
screen display, 24
New Executable (NE), 19
file format, 103
producing, 25
New Executable(NE)
examining, 104
null modem cable, 37

OEM ID, 110
OSFIXUP, 89

PAD, 14, 16
padding PME.SYS, 16
paging, 35
Paradigm compiler, 4
PARSECMDLINE, 10, 13
partitions, 20
PCCONF.H, 21, 63
PE loader, 22, 23
screen display, 24
PIC, 100
configuration, 46, 53
picsPM, 46, 54
PME.SYS, 15
PME22.TXT, 19
pmEasy internals, 51
pmEasy16 vs pmEasy32, 4
pmeVersionNum(), 73
pmiAllocLD(), 73
pmiAllocLDTop(), 73
pmiAllocMem(), 74
pmiAllocMemTop(), 74
pmiBaseToHandle(), 75
pmiCreateAlias, 75
pmiFreeLD(), 75
pmiFreeMem(), 76
pmiGetBaseAddr, 76
pmiGetDescr, 76
pmiGetIntDescr(), 76
pmiGetPMIntDescr(), 34
pmiGetPMIntVect(), 34, 77
pmiHeapStats, 77
pmiMapRealSeg(), 77
pmInit, 40
pmiSetBaseAddr(), 78
pmiSetExcepVect(), 78
pmiSetIntDescr(), 79
pmiSetLimit(), 79
pmiSetPMIntDescr(), 34
pmiSetPMIntVect(), 34, 80
pmiSetRights(), 80
pmiSimRealInt(), 83
pmiSizeMem(), 83
pmMain, 46
pmRestart, 40
Portable Executable, 19
section alignment, 97
Portable Executable (PE), 19, 35
binding, 106
examining, 105
file format, 104
producing, 25
producing, 25
sections, 104
preferred base, 23
protected mode, 3
entering, 45
running in, 46
switch to, 46
Protosystem, 1, 12
pseudodescriptor, 48
PSP, 11, 13

real mode interrupt, 83
real mode stub, 11
REL directory, 1
release notes, 19
release version, 27
restarting pmEasy, 16
restarting the application, 40, 62
return codes, 95
dpmiInit, 89, 95
loader, 95
return to DOS, 48
rmReturn, 48
ROM
loading from, 22
ROM application, 26
RTFS, 20
running pmEasy, 9
from boot disk, 14

from DOS, 9
from ROM, 16

sample applications, 1, 4, 27
building, 27
defines used in makefiles, 29
running, 28
screen display,disabling, 9
segments for flat mode, 34
services provided, 3
settings
A20 address line, 60
application name, 66
binding to application, 61
bootloading, 61
buffer area, 61
cross reference, 59
debugger, 61
debugging, 62
descriptor tables, 62
diagnostics, 62
disk reader, 63
disk reader drivers, 64
DPMI services, 64
heap, 64
keyboard, 66
loader and loading, 66
running pmEasy from ROM, 67
stack, 67
target BIOS/DOS, 68
video output, 68
shell isr, 32
small model, 54
SMX, 1, 4
smxDLM, 22, 23
smxFile, 109
Soft-Scope, 4, 7, 37, 47, 57
directions, 39
limitations, 41
restarting the application, 40
tips, 40
source code, 55
drivers, 55
SS == DS, 54
stack, 67
overflow/underflow, 22
protected mode, 67
real mode, 68
segment, 54
usage, 22
startup code, 30
stub program, 11, 12, 106
STUB statement, 12
summary of features, 3

TDUMP, 55, 104, 105
tiled descriptors, 35
tips, 97
tools, 7
for building pmEasy, 7
for building the application, 7
TR, 46
transferring app to target, 13
trap gate, 31
troubleshooting, 99
TSS0, 46

unDOS, 91
uninitialized variables, clearing, 34
utilities, 15
BOOTWRIT, 14, 15, 107, 110
CSICFG, 38, 61
DUMPBIN, 105
EXE2BIN, 14, 15
EXEHDR, 104
INTERLNK, 14
PAD, 14, 16
TDUMP, 55, 104, 105

version number, 73
video output, disabling, 9, 17
virtual 8086 mode, 4

Win32 emulation, 31
WIN87EM.DLL, 89
Windows, 4
Windows 9x MS-DOS command prompt, running from, 9
Windows NT, 16
running from, 10
writing code, 27

Zip disk, 19