Paradigm LOCATE

Version 6.0

Reference Manual

Paradigm Systems

Version 6.0
December 21, 1999
Manual version 6.0
PDF version 6.01

Paradigm Systems
Suite 2214
3301 Country Club Road
Endwell, NY 13760

(607)748-5966
(607)748-5968 (FAX)

Sales information: info@devtools.com
Technical support: support@devtools.com
Web: http://www.devtools.com
FTP: ftp://ftp.devtools.com

Please note that free technical support is available to registered users of the current release of any Paradigm software development tool. For prompt attention to your technical questions, please contact our tech support team via the Internet at *support@devtools.com.*

C O N T E N T S

I N T R O D U C T I O N

Paradigm LOCATE is a professional utility for preparing 16-bit Borland C++, Microsoft C/C++, and Paradigm C++ applications for use in embedded systems. Paradigm LOCATE is fast, easy-to-use, and creates the exact output files you need to develop and debug embedded system applications for the Intel, AMD or NEC x86-compatible microprocessors.

Paradigm LOCATE is unique in its support of all the Borland, Microsoft, and Paradigm software development tools, including all versions of their popular C, C++, and assembly language packages. With output file formats supporting the award-winning Paradigm DEBUG, popular in-circuit emulators and EPROM programmers, everything is included to help you get the most out of your embedded system application.

What's in Paradigm LOCATE

Chapter 1 tells you how to install Paradigm LOCATE. This introduction tells you about the features of Paradigm LOCATE.

Paradigm LOCATE, coupled with your favorite Borland, Microsoft or Paradigm compiler, assembler, and linker, is a complete embedded system development package. Just look at some of the many features offered by Paradigm LOCATE:

- **Fast:** No other product even comes close to Paradigm LOCATE in getting your application fully debugged and into EPROM.
- **Full Borland, Microsoft, and Paradigm support:** Use your favorite Borland, Microsoft, or Paradigm compiler to get the job done. Unlike other solutions, Paradigm LOCATE doesn't limit you to a single compiler vendor or a single version of a compiler.
- **C, C++, and assembly language:** Develop your application in the language of your choosing, knowing it is fully supported.

- **Startup code and run-time library support:** Paradigm LOCATE includes complete ROMable startup code for each supported compiler. Comprehensive run-time library support is also included for all memory models so you can use stream I/O, dynamic memory management, and floating point run-times in any embedded system without DOS or a BIOS.
- **Sample applications:** Plenty of sample applications, complete with makefiles and Paradigm LOCATE configuration files, are available to demonstrate various embedded system development techniques for each supported compiler.
- **Paradigm DEBUG support:** Choose from this award-winning family of source level debuggers that support stand-alone and in-circuit emulator debugging using any Borland, Microsoft, or Paradigm compiler.
- **Intel OMF86 support:** Absolute OMF86 output files with full debug information are available for users having an in-circuit emulator accepting this file format.
- **EPROM programmer support:** Paradigm LOCATE supports all popular EPROM programmer file formats, including Intel extended hex, binary and Tektronix hex. Paradigm LOCATE will also optionally split EPROM images or fill with any background pattern. Optimally-sized binary file output is also supported.
- **Compressed initialized data:** Constant and initialized data can be compressed in EPROM and decompressed by the startup code to save valuable EPROM space.
- **Configuration files:** A configuration file is how you inform Paradigm LOCATE about your target system address space, output file types, and other options. A full C preprocessor is standard, with macros, include files, and conditionals available to meet the most demanding requirements.
- **Chip select, wait state, DRAM refresh initialization:** Only Paradigm LOCATE can automatically generate processor-specific initialization code so there is no need to write custom startup code.
- **Reset vector initialization:** Paradigm LOCATE will, at your request, automatically create a far jump to the program entry point from the reset vector.
- **Stack initialization:** Stack initialization is also available, for applications which require a stack be setup automatically.

- **Target system documentation:** Create list files with any of the following information: segments, regions, public symbols, local symbols, or line numbers. Full application documentation is standard with Paradigm LOCATE.
- **Checksums and CRCs:** Calculate a PC ROM BIOS extension checksums or generate CRC16 or CRC32 checksums on any region of memory.

New features and changes

Some of the new features in Paradigm LOCATE version 6.0 are:

- Full support for Borland C++ 5.0 including integrated IDE support as well as new ANSI/ISO C++ language support for namesakes and the new keywords *bool* and *mutable*
- New support for the AMD Am186/188EM/ES/ER/ED embedded microprocessors
- Improved debug information support for Microsoft C/C++ compilers
- Continued support for all versions of Borland and Microsoft compilers from Turbo C 2.0 to Borland C++ 5.0 and from Microsoft C 5.1 to Visual C++ 1.52
- Full support for RTOS-aware Paradigm DEBUG, with enhanced support for all previous versions

Changed in version 6.0

- FILENAME option of ABSFILE, HEXFILE and LISTFILE directives now expects slashes (/) instead of backslashes (\ ) for pathname separators
- Checksums can use class names as well as physical addresses

Hardware/software requirements

Paradigm LOCATE runs on the IBM PC and compatible computers using a 486, or later, microprocessor and having a minimum of 8MB of memory. MS-DOS 6.2, Windows 3.1, Windows 95 or Windows NT is also required.

Paradigm LOCATE also requires a Borland, Microsoft, or Paradigm compiler and an assembler to build the compiler startup code and run-time library support packages.

The Paradigm LOCATE package

Your Paradigm LOCATE package consists of a diskette and this manual. The diskette contains all the programs and files you need to create embedded applications using the supported Borland and Microsoft compilers. The disk also contains sample applications demonstrating the use of the run-time libraries and Paradigm DEBUG.

The Reference Manual

The *Reference Manual* introduces you to Paradigm LOCATE and contains all the information needed to create embedded system applications with Borland and Microsoft compilers. This manual is arranged so you can either follow a short tutorial to quickly get up to speed or use it as a reference, depending on your level of experience.

Here are the key chapters in this manual:

- **Introduction:** introduces you to the key features of Paradigm LOCATE and tells you how to access the Paradigm technical support system.
- **Chapter 1: Installing Paradigm LOCATE** tells how to install Paradigm LOCATE on your system and how to create ROMable run-time libraries for your target system.
- **Chapter 2: Paradigm LOCATE basics** is a short tutorial of a simple embedded application built with Paradigm LOCATE.
- **Chapter 3: Relocation primer** is a review of the techniques used by Paradigm LOCATE to bind physical addresses to your segments.
- **Chapter 4: Using configuration files** is a detailed introduction into designing a custom Paradigm LOCATE configuration file for your target system.
- **Chapter 5: Configuration file directives** is the detailed review of the Paradigm LOCATE configuration file directives.
- **Chapter 6: Command line options** is the detailed review of the command line options available to Paradigm LOCATE users.

- **Chapter 9: Borland C++ guide** is a description of the Borland C++ compiler support package supplied with Paradigm LOCATE. Look here for specific tips and techniques for using Borland C++ with embedded systems.
- **Chapter 10: Microsoft C/C++ guide** is a description of the Microsoft C/C++ compiler support package supplied with Paradigm LOCATE. Look here for specific tips and techniques for using Microsoft C/C++ with embedded systems.

Also included in the Reference Manual are the following appendices. These contain useful information covering the use of Paradigm LOCATE and utilities.

- **Appendix A: Warning diagnostics** is a detailed description of the warnings output by Paradigm LOCATE.
- **Appendix B: Error diagnostics** is a detailed reference of all Paradigm LOCATE error messages.
- **Appendix C: Exit codes** lists the various exit code output by Paradigm LOCATE as a result of processing an input file.
- **Appendix D: INITCODE port definitions** is a list of supported peripheral register initializations supported by each processor.
- **Appendix E: AXE utility** is a short description of the Paradigm AXE file utility.
- **Appendix F: Hex file formats** documents the hex file formats supported by Paradigm LOCATE.

Wrapping up the manual is a comprehensive index, making all components of the Paradigm LOCATE *Reference Manual* available at your fingertips.

Technical assistance

Free technical support is available to registered users of the current release of any Paradigm software development tool.

If you have technical questions or need assistance in setting up or using Paradigm LOCATE, contact our technical support staff at (800)582-0864 during normal business hours (EST) or at (607)748-5966 if calling from outside North America. We will be more than happy to discuss your problem and provide the fastest possible response. Please have the following information available before you contact us:

- Product names and version numbers for all Paradigm products
- Product names and version number for third-party products, such as Borland C++ or Microsoft C/C++
- A detailed description of the problem, and how to reproduce it
- If sending us files, be sure to include a README file with the details of the problem, and your name, address, phone/fax numbers so we can get back to you. Please use a compression utility to keep the size of any files to a minimum.

The use of an on-line service is recommended since it offers timely turnaround of problem reports and maintenance releases of software.

We encourage all customers to contact us with their application, compiler, debugger, or in-circuit emulator questions. We have experts on staff to deal with any questions relating to Paradigm LOCATE, the use of Borland and Microsoft compilers in embedded systems, or using Paradigm DEBUG with an in-circuit emulator. Please feel free to contact us any time you need assistance.

E-mail

You may send technical questions or problem reports to our technical support group via the following e-mail address:

```
support@devtools.com
```

Internet

You can reach us on the Web at:

```
http://www.devtools.com
```

Internet users can access technical support, application notes, third party vendor information and product information on our website.

FTP

To obtain patch files, service packs and application notes quickly, access our anonymous FTP site at:

```
ftp://ftp.devtools.com
```

FAX

You may also fax your problem reports or questions to our technical support group at (607)748-5968. This is the least desirable method since we may lack the ability to reproduce your problem.

Problems and suggestions

We welcome your suggestions and feedback and hope you find that Paradigm LOCATE meets your requirements for embedded system software development. Paradigm LOCATE has been extensively tested prior to its release, but unforeseen problems or incompatibilities can arise due to the number of possible system configurations. Should you find a problem with this software or have an idea for an improvement, don't hesitate to contact us. We appreciate your feedback and suggestions for improving Paradigm LOCATE.

C H A P T E R

1

Installation

Paradigm LOCATE comes with an automatic installation and configuration utility called SETUP.EXE. SETUP.EXE will guide the installation process, setting up Paradigm LOCATE, installing the optional compiler support packages, and customizing ROMable run-time libraries for each memory model.

If you are not familiar with the Paradigm software license agreement, please take the time to read it now.

Now is a good time to make sure that you have filled in your product registration card; this guarantees that we will be able to keep you up-to-date about any new versions of Paradigm LOCATE and make you eligible for our free technical support.

This chapter contains the following information:

- installing Paradigm LOCATE on your system
- building ROMable libraries for your compiler
- accessing the README and FAQ files

Once you have installed Paradigm LOCATE, you'll be ready to create your own embedded system application. Refer to the Introduction chapter of this manual to learn more about which features of Paradigm LOCATE you will want to explore first.

Installing Paradigm LOCATE

A README file is included in the Information dialog of SETUP for answers to common questions.

To install Paradigm LOCATE from CD to your hard drive:

1. Insert the CD into your CDROM drive
2. Run SETUP.EXE from the CD
3. Follow the instructions on the screen

The **SETUP.EXE** program will open a series of Install options dialog boxes with the following selections for you to create a custom Paradigm LOCATE installation:

- Choose a compiler and version from the compiler support package (select this even if you are using assembly language - startup code is still the place to begin)
- Choose LOCATE compiler support options including compiler-specific example applications
- Choose to install ROMable run-time libraries for your compiler (select this if you are planning to use Borland or Microsoft C/C++ run-time libraries)

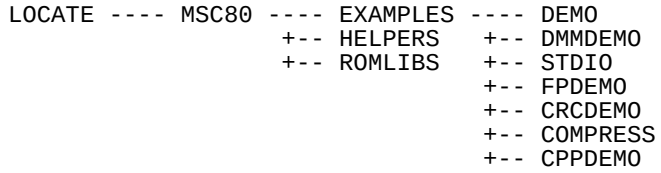
If the installation program is not compatible with your operating system, contact Paradigm for assistance.

The destination path defaults to c:\LOCATE, but you can specify a different drive and path if desired. It is recommended that you do not use long file names for your destination paths as some older development tools may not be compatible.

Your system will look slightly different, depending on your compiler.

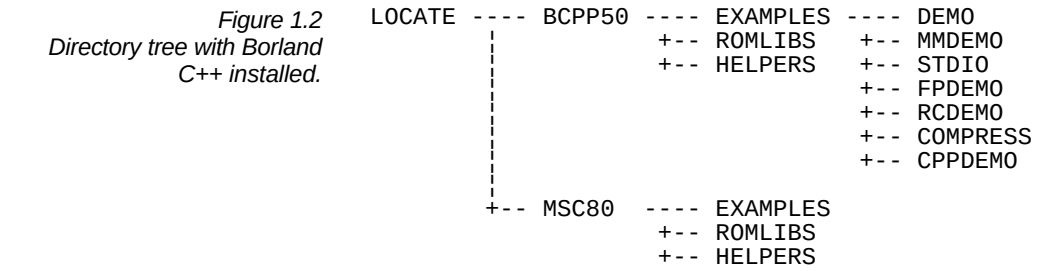
When the installation is complete, a directory tree structure similar to that in figure 1.1 will have been set up. The following is the organization of Paradigm LOCATE on your system after running SETUP with the default options and the Microsoft C/C++ 8.0 (Visual C++) compiler.

Figure 1.1 Sample Paradigm LOCATE directory tree



If you wish to install multiple compiler support packages, simply run the SETUP utility again for each compiler. For example, if we then installed the Borland C++ 5.0 support package startup code, the directory tree would look as shown in figure 1.2.

Figure 1.2
Directory tree with Borland C++ installed.



First time users should check out Chapter 2 to get familiar with the Paradigm LOCATE basics.

Following the installation session, go to the Paradigm LOCATE directory defined during the installation process and familiarize yourself with the compiler support package you will be using. Be sure to put the Paradigm software CD in a safe place in case it is needed in the future to add support for a different compiler.

Run-time library customization

The final step performed by the SETUP utility was to configure a subset of the compiler run-time libraries for use in an embedded system. This is an optional step and is only required if you plan to use the Borland/Microsoft run-time library routines in your embedded system. You can skip this section if SETUP correctly configured your run-time libraries.

As shipped by Borland and Microsoft, the run-time libraries contain many functions which are either not ROMable (functions which use self-modifying code or store data in a read-only segment) or are inappropriate for use in an embedded system, for example, functions which assume the presence of a file system.

By configuring a subset of the run-time libraries, potential incompatibilities involving the use of unsupported run-time library routines can be identified by the linker as an undefined external reference. With a custom run-time library configured, you won't find

yourself spending time debugging an application that should never have been built in the first place.

*Edit the makefile **MAKELIBS.MKF** and the library response files (the ***.LRF** files in the ROMLIBS directory) to create fully customized libraries.*

If you enabled the run-time library build option, the SETUP utility configured a single run-time library of your choosing that supports the example programs shipped with Paradigm LOCATE. If you later desire to change the makeup of a library or build different memory models, the DOS batch file **MAKELIBS.BAT**; (found in the ROMLIBS subdirectory of your compiler support package - see the directory tree on page 16) can be used to create ROMable run-time libraries that meet your specific requirements.

To build the ROMable libraries, the compiler, make and library utilities must be in the path.

The **MAKELIBS.BAT** batch file is very simple in its implementation. It invokes the **MAKELIBS.MKF** makefile (which is customized for the MAKE utility that came with the compiler) with one or more of the following parameters:

MODEL=*model* — This is the memory model of the library to be customized. Valid options are S, M, C, L, and H, depending on the compiler used.

SRC=*path* — This is the path to the run-time library to be customized, normally where you have installed your Borland or Microsoft compiler libraries.

DST=*path* — This is the destination path for the ROMable libraries. It is normally the same as the SRC macro definition but can be different if desired.

BIOS — Defined if you wish to include functions requiring an IBM PC-compatible BIOS. The default for this option is OFF.

EMU — Defined if you wish to include functions requiring the Paradigm run-time library helper functions. The default for this option is ON since the Paradigm LOCATE examples make extensive use of the run-time libraries.

CPP — Defined if you wish to include the C++ run-time

library functions (if supported by the selected compiler).

FLT This compiler-dependent macro is used to select the appropriate floating point model to be used with the embedded application.

> The **MAKELIBS.BAT** file always creates a copy of the run-time library with an 'R' prefix and never modifies the original library. For example, if you were building the Borland C++ small model libraries, the library CS.LIB would first be copied to the file RCS.LIB, from which the non-ROMable modules are removed.

Figure 1.3 is a sample **MAKELIBS.BAT** file modified to build the small and large run-time libraries for the Borland C++ compiler. Once the batch file completes execution, the files **RCS.LIB** and **RCL.LIB** will be left in the \BC\LIB directory. Because the macros BIOS and CPP are not defined, the BIOS and C++ run-time library support are stripped from the ROMable libraries.

*Figure 1.3 Sample **MAKELIBS.BAT** file*

```
echo off
make -fmakelibs.mkf -DMODEL=s -DSRC=\bc\lib -DDST=\bc\lib -DEMU
make -fmakelibs.mkf -DMODEL=l -DSRC=\bc\lib -DDST=\bc\lib -DEMU
```

The README file

A README file and Frequently Asked Questions (FAQ) file were installed by SETUP.EXE. The README contains last-minute information that may not be included in the manual.

If you did not get a chance to read the README file in the Information dialog during the installation process, you can view this file with any text editor or word processor. Please note any changes that might apply to your use of Paradigm LOCATE.

C H A P T E R

2

Paradigm LOCATE basics

Paradigm LOCATE includes everything you need to develop embedded system applications using your favorite Borland, Microsoft, or Paradigm compiler. Just some of the many features of Paradigm LOCATE include:

- support for all versions of Borland C++, Microsoft C/C++, and Paradigm C++
- comprehensive run-time library support, including floating point arithmetic, dynamic memory management, stream I/O, and C++ exception handling
- complete, working sample applications ready for debugging or burning into EPROM
- output file formats suitable for use with Paradigm DEBUG, in-circuit emulators and EPROM programmers
- full control of the target system address space, with exhaustive checking for potential problems

Tutorial

This section covers the use of Paradigm LOCATE in the embedded system software development cycle.

What better way to get started with Paradigm LOCATE than to work through a real application. This section illustrates how a Paradigm LOCATE configuration file is used to instruct Paradigm LOCATE to turn an application into ROMable format. Eventually, EPROMs are burned and the target system is up and running.

In presenting Paradigm LOCATE, the sample application has been compiled and linked with the Borland C++ compiler, so that you can focus on the use of Paradigm LOCATE. If you happen to be using a different compiler or a different version of the above compiler, the basic steps remain the same; they just happen to be implemented slightly differently.

Files in the tutorial

These files are located in the \LOCATE\DEMO subdirectory of the Paradigm LOCATE distribution diskette.

The tutorial is composed of a number of files:

SIEVE.C	The sieve application source code
SIEVE.ROM	Relocatable load module created by linker
SIEVE.MAP	Segment map created by linker
SIEVE.CFG	Paradigm LOCATE configuration file

The .ROM file may seem unfamiliar to most of you, however it is actually the .EXE file created by the linker. We use the .ROM extension because the relocatable module which is created for embedded systems is not truly executable under DOS or Windows, but has the same file format found in a DOS .EXE file. Since the Paradigm embedded startup code has been linked-in, running the .EXE file would most certainly crash your PC. The name change to .ROM prevents this accident.

The SIEVE application

The source modules are pre-built so we can focus exclusively on the use of Paradigm LOCATE.

Our tutorial application is the well-known Sieve of Eratosthenes benchmark program which is reproduced in figure 2.1. Since the same C source code can be compiled with Borland C++, Microsoft C/C++ or Paradigm C++, we'll introduce the Sieve source code and ignore the difference between the use of the compilers.

Figure 2.1 Sieve of Eratosthenes source code

```
 1  /*
 2  //  A demonstration of a simple ROMable application using
 3  //  Paradigm LOCATE.  This code can be built with either
 4  //  Borland C++ or Microsoft C/C++.
 5  */
 6
 7  #define  SIZE    1023/* Numbers to search for primes */
 8
 9  typedef  enum   {
10              FALSE,   /* Number is not a prime */
```

```
11             TRUE     /* Number is a prime */
12           } BOOL ;    /* Enums are better for debugging */
13
14  BOOL   flags[SIZE+1] ;     /* The array of BOOL flags */
15
16  void      main(void)
17  {
18     unsigned int   i, k, count ;
19
20     /* Loop forever once started */
21     for (;;)   {
22        /* Set all numbers to prime before starting */
23        for (count = i = 0; i <= SIZE; i++)
24           flags[i] = TRUE ;
25
26        /* Run the sieve and remove the non-primes */
27        for (i = 2; i <= SIZE; i++)   {
28           if (flags[i])   {
29              /* Cancel out all multiples of this prime */
30              for (k = i + i; k <= SIZE; k += i)
31                 flags[k] = FALSE ;
32              count++ ;
33           }
34        }
35     }
36  }
```

Since our purpose is to demonstrate the main features of Paradigm LOCATE, the actual source code is not useful, unless you have access to an in-circuit emulator. What is important is to see how Paradigm LOCATE uses a *configuration file* to find the instructions for mapping the application to the target system address space. Once you understand these simple rules, you will be ready to move on to the more advanced examples included in the compiler support packages.

The LOCATE configuration file

The key to using Paradigm LOCATE is the flexibility of the *configuration file*. It is not necessary to define any Paradigm LOCATE command line arguments. This means the configuration file completely defines the processing of the application.

Configuration file analysis

Let's take a detailed look at each of the Paradigm LOCATE configuration file directives in figure 2.2 to see their effect on the input and output files.

Only rarely must a segment be bound to a physical address.

When you study the configuration file, you will notice that *classes* are used to bind physical addresses to the segments in the application. This is done to avoid the problem that the number of segments in an application can be unbounded, yet the number of classes is almost always constant. By using a class as the basis for address binding, we don't have to deal with a large number of ungainly segments. Besides, we want to hide the implementation of physical segments from the Paradigm LOCATE user and instead focus on getting the application burned into EPROM.

Figure 2.2 Paradigm LOCATE configuration file

```
 1  //
 2  //  Configuration file for the example from the
 3  //  Paradigm LOCATE manual.  This is about as basic
 4  //  as an embedded application can get.  Check out the
 5  //  other examples for more sophisticated applications.
 6  //
 7
 8
 9  hexfile   intel86        // Intel extended hex for EPROMs
10  absfile   axe86          // Optional Paradigm DEBUG support
11  listfile  segments       // Segment map for documentation
12
13
14  //
15  //  Define how the target system address space is
16  //  partitioned.  Paradigm LOCATE will check that this
17  //  mapping is followed.
18  //
19
20  map  0x00000 to 0x0ffff as rdwr
21  map  0x10000 to 0xf7fff as reserved
22  map  0xf8000 to 0xfffff as rdonly
23
24  cputype  i80C186EA            // Target CPU is defined here
25
26  initcode reset              \ // Reset vector
27           umcs = 0xf838      \ // 32KB EPROM
28           lmcs = 0x0ff8        // 64KB RAM
29
30  dup      DATA ROMDATA         // Dup initialized data
31
32  class    CODE = 0xf800        // Code at F8000H
33  class    DATA = 0x0040        // Data at 00400H
```

```
34  class    ??LOCATE = 0xfff0    // 80C186EA chip select code
35
36  order    DATA                \ // RAM class organization
37           BSS BSSEND          \
38           STACK
39
40  order    CODE                \ // EPROM class organization
41           INITDATA EXITDATA   \
42           ROMDATA ENDROMDATA
43
44  output   CODE                \ // Classes containing code
45           INITDATA EXITDATA   \
46           ROMDATA ENDROMDATA  \
47           ??LOCATE
```

Let's take a detailed look at each line of the configuration file and see just what is going on here:

Lines 1-6 These are comments, so you can document what your configuration file is doing and why it needs to be done. Comments may be used freely throughout the configuration file.

Line 9 The HEXFILE directive is used to create **SIEVE.HEX**, an Intel extended hex file containing our sample application. This file can be downloaded to an EPROM programmer for preparing a set of EPROMs for our target system.

Line 10 The ABSFILE directive is used to create the file **SIEVE.AXE**, an absolute load module complete with debugging information. Paradigm LOCATE can create both Paradigm DEBUG and Intel OMF86 absolute load modules, ready to download and debug with an in-circuit emulator or Paradigm DEBUG.

Line 11 The LISTFILE directive creates an absolute segment map in the **SIEVE.LOC** listing file. These are the addresses where the application will appear in the memory address space of the target system.

Other information, like public symbols, can also be placed in this file (see page 70.

Lines 20-22 The MAP directives partition the target system memory address space into three mutually exclusive regions. The first MAP directive covers the 64KB of RAM found at the bottom of the memory address space while the third defines 32KB of EPROM at the top of the memory

address space. The second MAP directive marks as reserved the remainder of the address space. Paradigm LOCATE uses this information to warn if any code or data accidentally ends up in undefined regions of the memory address space, or overlaps multiple regions.

Line 24 The CPUTYPE directive identifies the target microprocessor as an Intel 80C186EA. This will permit the 80C186EA peripheral registers to be referenced in the INITCODE directive that follows.

Lines 26-28 Line 26 instructs Paradigm LOCATE to create a reset vector at address FFFF0H so control will be transferred to the application entry point when reset is asserted. Lines 27 and 28 are used to create initialization code for the 80C186EA upper and lower memory chip selects so the target system memory can be completely accessed.

*Paradigm LOCATE places this code in the class **??LOCATE**. See INITCODE, page 67 for more information.*

Line 30 The DUPLICATE directive makes a copy of the segments in the class DATA, which contain the initialized data. The compiler startup code then copies the contents of the EPROM-based ROMDATA class to the RAM-based DATA class.

Lines 32-34 The CLASS directive is used to bind physical segments to an application. The first CLASS directive places the program code at the base of the EPROM while the second CLASS directive puts the application read/write data immediately following the interrupt vector table. The third CLASS directive places the chip select initialization code we created in lines 27 and 28 at address FFF00H, which is within the 1KB block of addresses that the 80C186EA upper memory chip select can address following reset.

*If your target CPU does not require port initialization (lines 27 and 28), reference to **??LOCATE** can be eliminated.*

Line 36-38 Using DATA as the anchor class, this ORDER directive binds addresses to the other classes that are part of the RAM address space.

Lines 40-42 Using CODE as the anchor class, this ORDER directive binds addresses to the other classes that are in the EPROM address space. Notice that the copy of the initialized data in class ROMDATA is placed in the EPROM address space where it will be copied to RAM by the startup code.

Lines 44-47 The OUTPUT directive identifies which classes should be placed in the output file. Note that classes containing code or copies of initialized data are required to be named in the OUTPUT directive. Other classes, containing uninitialized data and the state, can be left out since they are initialized by the application.

Running Paradigm LOCATE

Now that we understand the basics, it's time to try out Paradigm LOCATE and get our first embedded application ready for burning EPROMs or debugging with Paradigm DEBUG.

When Paradigm LOCATE is started, it reads in the **SIEVE.ROM**, **SIEVE.MAP**, and **SIEVE.CFG** files to build the stand-alone embedded application. This can be as simple as the following line demonstrates:

```
locate sieve
```

When you run Paradigm LOCATE, the instructions in the configuration file **SIEVE.CFG** direct the creation of the Intel extended hex file containing the code and data at the specified addresses, along with an AXE file suitable for use with Paradigm DEBUG and an in-circuit emulator. An examination of the physical segment map from the file **SIEVE.LOC** in figure 2.3 clearly shows the transformation to an absolutely addressed program, ready for your target system.

The last two segments in the segment map are the 80C186EA peripheral register initialization code and the reset vector, both created in the INITCODE directive on line 26. If you make any changes, such as changing the processor type or adding stack initialization code, your own results may be somewhat different. For example, the segment **??CPUINIT** is only present if you specified I/O operations with the INITCODE directive.

Figure 2.3
SIEVE.LOC file contents

```
1  Input file: SIEVE.ROM
2  Configuration file: SIEVE.CFG
3  Created by Paradigm LOCATE 5.11 on Tue Jun 11 10:04:14 1996
4  Command line options: C:\LOCATE\LOCATE.EXE sieve
5
```

```
 6  Memory Address Map for Program SIEVE
 7
 8   Start     Stop      Length   Segment           Class
 9   000400H   000403H   00004H   _DATA             DATA
10   000404H   000C03H   00800H   _BSS              BSS
11   000C04H   000C05H   00002H   _BSSEND           BSSEND
12   000C10H   00140FH   00800H   _STACK            STACK
13   0F8000H   0F8132H   00133H   _TEXT             CODE
14   0F8134H   0F8134H   00000H   _INIT_            INITDATA
15   0F8134H   0F8134H   00000H   _INITEND_         INITDATA
16   0F8134H   0F8134H   00000H   _EXIT_            EXITDATA
17   0F8134H   0F8143H   00010H   _EXITEND_         EXITDATA
18   0F8150H   0F8150H   00000H   _RD               ROMDATA
19   0F8150H   0F8153H   00004H   _DATA             ROMDATA
20   0F8160H   0F816FH   00010H   _ERD              ENDROMDATA
21   0FFF00H   0FFF12H   00013H   ??CPUINIT         ??LOCATE
22   0FFFF0H   0FFFF4H   00005H   ??BOOT            (ABSOLUTE)
23
24  Entry point:   FFFF:0000
25  Initial stack: 00C1:0800
```

Burning EPROMs

At this point, we have the file **SIEVE.HEX** which is ready to be burned into EPROM and installed in our target system. If you don't have an EPROM programmer that supports Intel extended hex, don't worry - a quick edit of line 9 can create an output file in either Intel hex, binary or Tektronix formats. Here is the replacement directive for creating an Intel hex output file which occupies the upper 32KB of the address space:

```
hexfile  intel80 offset=0xf8000
```

In this example, we must specify an offset since Intel hex format can contain at most 64KB of data. Since we have as much as 1MB of address space to deal with, the OFFSET option informs Paradigm LOCATE which 64KB of the 1MB address space to work with.

If you are using a system with a 16-bit external bus, you will need a pair of EPROMs, one containing the even addresses and one containing the odd addresses. The following HEXFILE directive can be used to create the files **SIEVE.HX0** (containing the code on even addresses) and **SIEVE.HX1** (containing the code on odd addresses) if your EPROM programmer can't perform the split for you.

```
hexfile  intel80 offset=0xf8000 split=2
```

Debugging options

So much can go wrong with embedded applications, and without the right tools, figuring out what went wrong can be downright frustrating. One area that Paradigm LOCATE excels is in providing the most popular output file formats for debugging, using the award-winning Paradigm DEBUG or an in-circuit emulator accepting the standard Intel OMF86 file format.

Paradigm DEBUG/RT users would need to make some minor changes to the configuration file.

Our sample configuration file already set us up for a debugging session with Paradigm DEBUG (used in conjunction with a supported in-circuit emulator), so there is nothing left to chance. Just fire up the hardware, run Paradigm DEBUG and have the file **SIEVE.AXE** downloaded for an instant debugging session.

If you don't have access to Paradigm DEBUG but have an in-circuit emulator supporting the Intel OMF86 standard, changing output file formats is as simple as changing the ABSFILE option on line 10 from AXE86 to OMF86. Instead of the AXE86 output file, Paradigm LOCATE will create an Intel OMF86 file, ready to download to your emulator.

Summary

If you haven't done so, now is a good time to try some the examples included in the compiler support packages. See chapters 9 or 10 for further information.

This completes our look at a simple embedded application and how Paradigm LOCATE contributed to preparing files for debugging or burning EPROMs. By applying the techniques we discussed while progressing though the different examples, you are well on the way to becoming fluent with the capabilities of your compiler and completing your own embedded system application.

Now that you are a little more comfortable using Paradigm LOCATE, you might want to continue on and check out the more sophisticated compiler-specific applications like C++, floating point, dynamic memory management, or stream I/O that are part of each compiler support package. Each example has a makefile, complete with options for setting the memory model and debugging options. This makes it simple to customize each example to meet your own requirements.

C H A P T E R

3

Relocation primer

This section contains optional information provided for those interested in the segment relocation process, handling of initialized data, and other topics of interest to embedded system programmers. Paradigm LOCATE can be used quite well without understanding these underlying algorithms, so this section may be skipped at the discretion of the user.

Relocation basics

A segment is the basic unit of organization.

When a linker processes a set of object files, it combines all *segments* having the same segment name into a single physical segment which must fit within a 64KB region of the memory address space.

A class is a collection of related segments.

Compilers typically assign each segment to a *class*, and assembly language users can do the same. Assignment to a class permits the linker to combine together similar segments, such as all segments containing code or initialized data, so they can be manipulated together as a single entity. Although a member of the class, each segment remains independently addressable and can vary in length to a maximum of 64KB bytes. Since any number of segments can form a class, there is no restriction on the size of a class.

We will cover how groups affect the relocation process on page 36.

Compilers and assemblers also define a different relationship between segments known as a *group*. The segments within a group do not

have to be contiguous but are all addressed using the same segment base; they must fit into a single 64KB physical segment. When the linker encounters a group, it replaces the offsets from the segment base with offsets from the group base, adjusting them upward as necessary.

The linker output files

The linker has the responsibility of resolving all external references and creating the relocation table containing the list of segment fixups. Although all external references have been resolved, the segment fixups are still relocatable and can be moved anywhere within the 1MB address space, which is where Paradigm LOCATE becomes involved.

The .ROM and .EXE files are really the same - we just want to distinguish them.

This information, along with other loading instructions and optional debugging information, is written out in the .ROM and .MAP files. The .ROM file is the relocatable load module. By default, linker names in this file have an .EXE extension. The .MAP file is the segment map file. Both .ROM and .MAP files are required by Paradigm LOCATE. Being a relocatable load module has certain advantages and is a necessary requirement for DOS, since the final physical addresses of a program are unknown until the program is loaded.

Of course, for designers of embedded systems, this is unacceptable since all segments must be at fixed addresses before the code is committed to EPROM or downloaded to an in-circuit emulator. A utility like Paradigm LOCATE solves this problem by extracting the segments and relocation information from the linker files and converting the relocatable segment references to absolute addresses in the target system address space, as directed by the configuration file.

Segment aliases

Paradigm LOCATE will automatically warn of alias conditions.

The *virtual segment*, or frame number, is used as a handle by Paradigm LOCATE to identify the target segment referred to by a fixup record.

It turns out that it is possible for two segments to share the same virtual segment number, a situation known as aliasing. Since the fixup records for aliased segments are indistinguishable, some restrictions are placed on the developer to prevent aliases from being created.

Segment aliases occur when a segment fails to cross a paragraph boundary, and the start of the second segment shares the same virtual fixup as the first. Whether or not a segment alias presents a problem depends on whether the segments are members of the same class. If both segments are members of the same class there is never a problem since these segments will be located contiguously and the fixup is unambiguous. If the offending segments are organized as a group, there is again no problem since all segments in a group share a common virtual segment number and the segment fixup will also be unambiguous.

The segment alias problem arises when the segments are members of different classes and an attempt is being made to relocate the segments to different regions of the memory address space by splitting them. If a segment fixup is requested for an aliased virtual segment, the fixup is ambiguous and Paradigm LOCATE cannot determine the correct address translation.

The startup code supplied with Paradigm LOCATE will always prevent an alias, unless you modify it.

Fixing a segment alias is generally easy since a segment alias condition can only occur when the length of the first segment and the alignment of the next segment in the load module result in both segments having segment bases in the same paragraph. Since the alias is a function of segment length and alignment, adjusting either of these two parameters can eliminate the possibility of a segment alias occurring.

Segment ordering and alignment

The solution to the segment alias problem involves specifying the alignment characteristics for the first segment of each class such that the start of the segment will be forced to the new paragraph.

This is easily accomplished by using the assembly language startup code to declare the segment alignment of the first segment in a class to be on a paragraph boundary. This will allow the startup code to take

advantage of the way the linker organizes segments and classes within the load module.

This is the rationale behind the declaration of the first segment in each class before any code or data declarations.

The ability to control the segment length is limited, especially when high level languages or pre-compiled library modules are involved. We have seen that the DOS linkers order and align the segments in the load module in the order they are encountered in the object modules. By making sure that the first object file input to the linker specifies the desired segment order and alignment for all the classes in the application, the user has complete control over the final ordering and alignment of the segments in the load module.

These are only an example - use the DefSeg macros supplied with the startup code whenever possible.

The following are sample declarations which demonstrate the technique. Notice that the first segment in each class has been declared to be paragraph aligned using the assembler keyword 'para'. So long as the previous class is not empty, this will guarantee a unique segment address for the class. Also note that the subsequent segments in a class can use any alignment since they are always manipulated together and never split apart:

```
_TEXT    segment para   public   'CODE'
_TEXT    ends
_DATA    segment para   public   'DATA'
_DATA    ends
```

The case where a segment will have zero length, yet must be manipulated independently, will be examined in the next topic, where it arises naturally.

Segment checking

After converting from virtual to physical segment addresses, Paradigm LOCATE checks for overlapping segments and outputs a warning if any are detected.

A segment overlap warning is generally the result of the class length increasing to the point where it overlaps with one or more of the following classes. This problem is easily corrected by changing the starting addresses in the configuration file CLASS directives to match the physical memory requirements of each class.

Also checked by Paradigm LOCATE is data exceeding the upper limit of the CPU memory address space. This condition would occur if the sum of the segment base address and the length of the segment exceeds the 20-bit addressing capability of the microprocessor. This is not an uncommon problem, as it is quite easy for the application code to grow past the 1MB boundary.

Any attempt to use the reserved address is flagged by Paradigm LOCATE.

Another possibility that Paradigm LOCATE will check for is an application completely filling the RAM or ROM address space assigned to it. To check for code or data spilling into non-existent regions of the address space, Paradigm LOCATE permits the user to define regions of the memory address space which are reserved and cannot be used.

Absolute segments

*If necessary, the warning messages for absolute symbols can be disabled with the **-w-** command line option.*

A potential problem with the use of DOS linkers is that segments declared at an absolute address do not appear in the output link map. Since any symbols defined in an absolute segment will appear as part of the debugging record, an attempt by Paradigm LOCATE to convert the virtual segment to a physical segment address will most likely fail. When the virtual to physical segment translation fails, Paradigm LOCATE assumes that the symbol is a member of an absolute segment and does not fixup the segment component of the code or data and issues a warning.

Absolute segments are also very limited in that they can only be used to define addresses.

The use of absolute segments is not recommended since Paradigm LOCATE allows the user to delay the binding of the physical segment address until the locate phase rather than when the file is assembled. Besides leading to more portable code, error checking is enhanced since Paradigm LOCATE can confirm that no other segments will overlap the absolute segment.

There is also the possibility that one of the other segments in the application will have a logical segment index identical to the absolute segment. Since Paradigm LOCATE has no way to verify the symbol being absolute, the translation would take place and the address of the symbol in the debugging records would be incorrect. While a problem for the debugging information, this event would not affect the correctness of the code.

Fixing absolute segments

The two steps needed when converting an application from using absolute segment addressing to using Paradigm LOCATE to fix the segment address are shown below. The first step is to change the segment declaration in the assembly language source from the absolute format to the relocatable format with a unique class name.

```
ASEG segment at 0f000h              ; Absolute
; your code
ASEG ends

ASEG segment para public 'MYSEG'    ; Relocatable
; your code
ASEG ends
```

The second step involves adding a directive to the Paradigm LOCATE configuration file to set the base address of the segment to the original segment address.

```
class MYSEG = 0xf000     // Fix the address
```

Groups

Languages such as Microsoft C/C++ and Borland C++ use the group DGROUP.

Currently there is no explicit support for groups in Paradigm LOCATE due to DOS linkers lacking sufficient information on the segments that make up a group.

If you are programming in assembly language, this should not cause any problems since the groups and classes used are controlled completely by the programmer. C and C++ application programmers should pay careful attention to make sure that the rules for manipulating groups are not violated.

The order of classes in a group must follow the ordering the .MAP file.

Paradigm LOCATE provides support for groups through the use of the configuration file ORDER directive. After processing the object modules, the linker adjusts the offsets within each segment in a group relative to the start of the group. If the user supplies the class name of the first class in a group, the other classes in the group can be relocated relative to the base segment of the group. The location of a group is handled by assigning the first class in the group a physical segment with the CLASS directive and ordering the remaining classes in the group located with the ORDER directive.

Duplicating classes

Some programs define initial values for read/write data structures that are assumed to be correct when a program begins execution. Since this is not the default case for a system just powered up, Paradigm LOCATE must provide a mechanism for initializing this memory to its initial values.

You can't compute the size of a duplicated class, only of the original class.

The startup code is responsible for the initialization of RAM-based data from an EPROM-based copy. This technique involves the creation of a placeholder class which has a segment address but has no length since the actual segments in the class will be filled in by the Paradigm LOCATE DUPLICATE directive. Since the placeholder class will have zero length, any class that follows is guaranteed to be aliased.

See your compiler startup code for an example of this technique.

The solution to this problem is to define a pair of classes, the first serving as the placeholder and the second serving to mark the end of the first.

```
_rd      segment para   public   'ROMDATA'
ridata   label   byte
_rd      ends

erd     segment para   public   'ENDROMDATA'
        db  16 dup (?)
_erd    ends
```

The above segment declarations define both classes to be paragraph aligned with the second class following the first. While we cannot avoid the alias condition, we can make it harmless by making sure that the second class is always located contiguously to the first. The Paradigm LOCATE ORDER directive can now be used to fix the relationship of the classes, relative to an anchoring class.

```
order  CODE ROMDATA ENDROMDATA
```

Finding the start of the class ROMDATA is as simple as taking the address of the label `ridata` or referencing the segment name. The end of the class is marked by the class ENDROMDATA, which also guarantees that the following class will not be aliased. This is determined by the length of the class being 16 bytes, guaranteeing the following class will have a unique fixup.

C H A P T E R

4

Using configuration files

The process of converting the relocatable output of the linker to a format suitable for downloading to a remote debugger, an in-circuit emulator, or an EPROM programmer begins with the instructions contained in a Paradigm LOCATE *configuration file*. A configuration file contains any number of *directives* which allow you to control where your application will reside within the target system memory address space, the number and type of the output files, and any other Paradigm LOCATE options of your choosing. Each directive may also accept *options* which provide more specific results for the directive.

Because Paradigm LOCATE configuration files use a C preprocessor, you have full control over the application with macros, conditional processing using standard C syntax.

Configuration file format

The default Paradigm LOCATE configuration file is the filename of the load module with the .CFG extension. For example, assuming you just linked your application and have the newly created files **DEMO.ROM** and **DEMO.MAP**, the following Paradigm LOCATE command line would use the default configuration file **DEMO.CFG**

for the directives to process the **DEMO.ROM** input file and create the requested output files.

```
locate demo
```

Later on, we will see how the preprocessor can help manage multiple configurations.

Often it is more convenient to use different configuration files as you proceed through the phases of the software development cycle or to have multiple projects share a common configuration file. Using the Paradigm LOCATE **-c** command line option, the default configuration filename can be overridden and a configuration file of your choosing substituted. Paradigm LOCATE also offers full control over the default file extensions. If you prefer to use a different configuration file extension on a project basis, the Paradigm LOCATE command line option **-Xc** can be placed in the **LOCATE.OPT** file to substitute your own default configuration file extension when Paradigm LOCATE is run.

Paradigm LOCATE gives you considerable leeway in the layout of your configuration file. With the exception of the few directives that depend on options specified in a previous directive, Paradigm LOCATE directives can be declared in any order in the configuration file.

Directive format

Here is the format of a typical configuration file directive processed by Paradigm LOCATE:

```
directive option [ option ... ]
```

Each configuration file directive accepts one or more *options* which customize the actions of the directive to meet specific requirements. Some directives accept a single option while others accept an unlimited number of options. When a directive accepts multiple options, the options can appear in any order unless otherwise specified.

For example, the LISTFILE directive is used to create an absolute listing file containing segments, publics, line numbers and local symbols. In the following example, both LISTFILE directives are equivalent.

```
listfile  segments publics lines
listfile  publics lines segments
```

Line continuation

Paradigm LOCATE processes each configuration file directive up to the end of the line. For readability, and to permit an arbitrary number of options in a directive, multiple physical lines can be combined into a single logical line using a line continuation character, the backslash (\). The following is a simple example of using line continuation in the WARNINGS directive, used to enable and disable specific warnings.

```
warnings  -w1000   \   // Turn off warning 1000
          -w1001   \   // Turn off warning 1001
          +w1002       // Turn on warning 1002
```

Note that in the previous example, all text following the line continuation character up to the end of the line is ignored. This permits comments to be added to the source line, or allows the formatting of the directive options in a vertical line. While the WARNINGS directive is just as happy having all the options listed on a single line, the line continuation feature of Paradigm LOCATE permits a clear view of the directive options without destroying the layout or readability of the configuration file.

Directive processing priority

Paradigm LOCATE options can be specified in the **LOCATE.OPT**; file, on the DOS command line or in the configuration file. In the event of conflicting options, the following processing order (lowest to highest) is used to determine which Paradigm LOCATE options will be enabled.

- **LOCATE.OPT** options
- configuration file directives
- command line options

With the exception of the **-c** command line option which is processed immediately, all other command line options are processed after the configuration file directives have been processed. This permits the command line to be used to override any default actions specified in the configuration file or in the **LOCATE.OPT** file.

In the event of multiple directives within the configuration file, subsequent directives will override the effect of the previous directives, except for instances of the HEXFILE and LISTFILE directives which always specify multiple, independent actions. Command line options which enable an action can be disabled later by the complementary command line option just as a later configuration file directive can enable or disable a previous directive.

The preprocessor

In order to accommodate a diverse range of options, a full C preprocessor is used to prepare configuration files for parsing by Paradigm LOCATE. The preprocessor gives you great power and flexibility in the following areas:

- Defining macros to reduce the development effort and improve the readability of your configuration files
- Including directives from other files, such peripheral device definitions
- Setting up conditional processing for improved portability and for managing multiple builds

Preprocessor directives can appear anywhere in the configuration file.

Any line with a leading # is taken as preprocessing directive. The initial # can be preceded or followed by white space if desired.

The #define directive

The **#define** directive defines a macro, with or without parameters, as shown in the following example:

```
#define macro_indentifier <token_sequence>
```

Each occurrence of *macro_identifier* in your configuration file will be replaced with the *token_sequence*, which may be empty.

The #undef directive

You can undefine a previously defined macro by using the #undef directive:

```
#undef macro_identifier
```

Once undefined, it can be redefined with **#define**, using the same or a different token sequence.

The -D option

The **-D** option can be used on the Paradigm LOCATE command line to define identifiers before the start of the configuration file processing.

The Paradigm LOCATE command line

```
locate -DDEBUG=1 -DLIST test
```

is equivalent to placing

```
#define   DEBUG   1
#define   LIST
```

at the start of the **TEST.CFG** configuration file.

File inclusion with #include

The **#include** directive is used to pull in other files into the original configuration file. It uses one of the following forms, which are treated the same by Paradigm LOCATE:

```
#include <filename>
#include "filename"
```

The action of the preprocessor is to remove the **#include** line from the configuration file and replace it with the entire contents of the file *filename*.

The #if, #elif, #else, and #endif directives

Paradigm LOCATE supports conditional processing using the **#if**, **#elif**, **#else**, and **#endif** directives. Using these directives you can conditionally include configuration file source lines, based on the result of an expression:

```
#if expression
<section>
#elif expression
<section>
#else
<section>
#endif
```

If an expression evaluates to non-zero (after any macro expansion), the source lines represented by the corresponding section are preprocessed and passed on to Paradigm LOCATE. When an expression evaluates to zero, the corresponding section is ignored.

The #ifdef and #ifndef directives

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is defined, that is, whether a previous **#define** is still in force for the identifier. The line

```
#ifdef identifier
```

has exactly the same result as

```
#if 1
```

if *identifier* is defined, and the same effect as

```
#if 0
```

if *identifier* is undefined.

#ifndef is used to test for the not defined condition.

The operator defined

The **defined** operator offers a more flexible method of testing whether one or more identifiers are defined. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) evaluates to 1 (true) if the identifier has been previously defined and has not been undefined, otherwise it evaluates to zero. The directive

```
#if defined(aSymbol)
```

is the same as

```
#ifdef aSymbol
```

The advantage of using the **defined** operator is that it can be used repeatedly in a complex expression, such as

```
#if defined(thisSymbol) && !defined(thatSymbol)
```

The #error directive

The **#error** directive is used to terminate processing and output an error diagnostic of your choosing. The **#error** directive is typically used in a conditional clause to catch an unexpected condition, as shown in the following example:

```
#if !defined(A_MACRO)
#error  Failed to define macro A_MACRO
#endif
```

Predefined macros

The following macros are predefined by Paradigm LOCATE for use in configuration files:

```
__PARADIGM__ 1
__LOCATE__   Paradigm LOCATE version number
```

Comments

Configuration files do more than instruct Paradigm LOCATE how to process the relocatable load module; they are also a key component of the design documentation. To help you in properly documenting your design, comments can be added freely to the configuration file.

Old-style C comments may also be used.

The start of a comment field is defined using the C++ syntax, which is a pair of slashes ('/') with the comment continuing to the end of the line. Blank lines and comments can appear anywhere within the configuration file, improving the readability while providing complete flexibility.

Finding errors

These diagnostics are designed to pinpoint errors or warn of hazardous conditions.

Any time Paradigm LOCATE finds a discrepancy parsing a configuration file directive, it issues a diagnostic indicating the configuration file source line in error. A complete list of diagnostics produced by Paradigm LOCATE, together with a description of the probable cause and possible corrective actions, can be found in Appendix A for warning diagnostics, and in Appendix B for error diagnostics.

Check both the reported line and the previous line for the error.

In the event of an error in a directive spanning multiple lines, the source line number reported in an error diagnostic may be inaccurate. Because the reported position may be the line following the actual error, it is important to examine the entire directive for the error, not just the reported line.

C H A P T E R

5

Configuration file directives

This chapter offers a detailed description of the Paradigm LOCATE *configuration file directives*, the commands that build an absolute load module in a file format of your choosing. Before introducing each directive, a short review of the layout used to document the directives is in order.

The Paradigm LOCATE configuration file directives contain a detailed description of the directive, the syntax and a list of options accepted by the directive. Any command line equivalent options are also listed to round out the detailed description. To place each directive in the context of an application, each entry concludes with a list of examples showing the directive as it might be used in a Paradigm LOCATE configuration file for a typical embedded system.

All user input is shown in lowercase

All configuration file directives are shown with the directive and any options shown in uppercase. A valid directive or option must have at least the significant characters although it may have more. Paradigm LOCATE keywords are case-insensitive so you can use either upper or lowercase in your configuration files. Options to directives are also case-insensitive, with the exception of segment and class names which are case-sensitive and must match the names from the link map.

Any optional arguments for a directive are shown enclosed by square brackets ([and]) with an ellipsis (...) used to indicate repeated arguments. The following mnemonics are used throughout to identify the type of argument expected by Paradigm LOCATE.

- *data* 8- or 16-bit data
- *data8* 8-bit data
- *data16* 16-bit data
- *addr16* 16-bit segment (paragraph) address
- *addr20* 20-bit physical address
- *addr24* 24-bit physical address
- *addr* A **addr20** or **addr24** address (depending on the input file)
- *file* A filename with optional path. A valid Paradigm LOCATE filename must begin with a letter and be followed by any combination of letters or numbers.
- *list* One or more class names
- *name* A segment or class name

ABSFILE

Description

The ABSFILE directive is used to select the file type and optionally supply a filename for the absolute output file. The ABSFILE directive is used when you will be working with Paradigm DEBUG, or a development tool accepting Intel OMF86 files.

Syntax

```
ABSFILE [ AXE86 | OMF86 | NONE ]   \
[ FORMAT=type ]                    \
[ FILENAME=file ]
```

Options

The following are valid options for the ABSFILE directive:

AXE86 Selects the Paradigm DEBUG 6.0 format for the absolute output file. The default file extension is .AXE and may be changed with the **-Xa** option.

OMF86 Selects the Intel OMF86 format for the absolute output file. The default file extension is .ABS and may be changed with the **-Xo** option.

NONE Disables the creation of an absolute output file.

FILENAME This argument permits you to change the name of the absolute output file to file. The default filename is the same as the input file with the extension determined by the output file type.

> Use slashes (/) instead of backslashes (\ ) for path name separators. For example,
> `file = c:/output/test.axe`

All current versions of Paradigm DEBUG use the default AXE file format.

FORMAT The FORMAT option is used to specify different AXE file formats, for use with older versions of Paradigm DEBUG. This option accepts a single argument, depending on the version of the debugger being used.

`PD60`	Paradigm DEBUG 6.0
`PD50`	Paradigm DEBUG 5.0
`PD40`	Paradigm DEBUG 4.0
`PD31`	Paradigm DEBUG 3.1

PD30	Paradigm DEBUG 3.0
PD20	Paradigm DEBUG 2.0
PD10	Paradigm DEBUG 1.0

Command line options

The following ABSFILE directives can be specified from the Paradigm LOCATE command line as well as in the configuration file.

-Apd60	AXE86 FORMAT=PD60
-Apd50	AXE86 FORMAT=PD50
-Apd40	AXE86 FORMAT=PD40
-Apd30	AXE86 FORMAT=PD30
-Apd20	AXE86 FORMAT=PD20
-Apd10	AXE86 FORMAT=PD10
-Aomf	OMF86
-Ad	NONE
-An*file*	FILENAME=*file*

Examples

```
absfile  omf86 filename=myprog.abs
absfile  axe86 format=pd40
```

Description The CHECKSUM directive is used to define a region of the memory address space and calculate the CRC or checksum of that region. At run-time, the target system can then compare the computed CRC or checksum with the stored value to determine if any changes have been made to the program or data.

Syntax

```
CHECKSUM    addr TO addr       \
[ ADDRESS=addr ]               \
[ FILL=fill ]                  \
[ ROMBIOS | CRC16 | CRC32 ]
```

Options The following are valid options for the CHECKSUM directive.

ADDRESS The ADDRESS option is used to set the physical address of the checksum. If not specified, the computed checksum will default to the address immediately following the end of the checksum region. The address can also be the name of a class, as well as a 20-bit physical address.

FILL The FILL option is used to inform Paradigm LOCATE of the background pattern of unused bytes within the checksum region. The value used for the fill and background contents of the EPROM must agree for checksum calculation to occur. If not specified, the fill pattern defaults to 0xFF.

ROMBIOS This option selects the IBM PC ROM BIOS extension checksum for the defined region, which writes a one byte checksum at the specified address.

CRC16 The CRC16 option selects the CRC-16 checksum for the defined region, which writes a two byte checksum at the specified address.

CHECKSUM

CRC32	This option selects the CRC-32 checksum for the defined region, which writes a four byte checksum at the specified address.

Command line options

None

Examples

```
checksum 0xc0000 to 0xc07fe fill=0xff rombios
checksum 0xf8000 to 0xffffd address=0xffffe crc16
checksum CODE to ROMDATA crc32
```

Description The CLASS directive is used to assign a physical address to each of the segments in a class.

Syntax `CLASS` *`classname = addr16`*

Options The 16-bit segment address in the argument *addr16* is bound to the first segment in the class *classname*. Each of the remaining segments in the class are then assigned physical addresses that are relative to preceding segments in the class.

Command line options None

Examples

```
class  CODE = 0xfc00
class  DATA = 0x0040
```

COMPRESS

Description

The COMPRESS directive is used to compress a duplicated class, reducing the size of the class to save space. Paradigm LOCATE will write out a compressed version of the named class in the output file.

A sample application demonstrating compression is included with each compiler.

Each Paradigm LOCATE compiler support package includes a decompression module that decompresses the result during the startup code. This module is automatically inserted into the ROMable run-time libraries.

Paradigm LOCATE runs a two step compression algorithm to compress a class. During the first phase, Paradigm LOCATE estimates the compressed size of the class, a requirement for the binding of addresses to the segments and classes that follow the compressed class. In the second phase, the class is compressed after any segment fixups have been applied.

Syntax

```
COMPRESS  classname
```

Options

classname is the name of the class to be compressed. This class must appear in a DUP directive as it is not possible to decompress in place.

Command line options

None

Examples

```
dup  FARDATA ROMFARDATA
compress ROMFARDATA
```

CPUTYPE

Description The CPUTYPE directive informs Paradigm LOCATE of the target system microprocessor. Paradigm LOCATE uses the CPUTYPE directive to select the set of peripheral registers permitted in the INITCODE directive.

Syntax `CPUTYPE` *`cpu_id`*

Options The following is a list of microprocessor IDs supported by the *cpu_id* argument.

```
I8088          D70108 (V20)
I8086          D70116 (V30)
I80188         D70208 (V40)
I80186         D70216 (V50)
I80C188        D70320 (V25)
I80C186        D70330 (V35)
I80C188EA      D70325 (V25+)
I80C186EA      D70335 (V35+)
I80L188EA      D70136 (V33)
I80L186EA      D70236 (V53)
I80C188EB      D70423 (V55SC)
I80C186EB      D70433 (V55PI)
I80L188EB      D70208H (V40H)
I80L186EB      D70216H (V50H)
I80C188EC
I80C186EC      AM186CC
I80C188XL      AM186EM
I80C186XL      AM188EM
I80286         AM186ES
I80386         AM188ES
I80386CX       AM186ER
I80386EX       AM188ER
I80486         AM186ED
```

Command line options None

CPUTYPE

Examples

```
cputype  i80c186eb
cputype  i80c188xl
cpu      D70325
```

Description

The DEBUG directive is used by Paradigm LOCATE to determine which debug information data structures will be included in the Intel OMF86 output file. By eliminating unnecessary debugging information such as types, Paradigm LOCATE can run significantly faster while producing smaller output files.

This directive is also used to enable Paradigm DEBUG OMF86 extensions or force compatibility with the Intel iC86 C compiler. These extensions add support for enumerations and C++ objects and are used by third-party debugging tools that accept OMF86 files.

Syntax

`DEBUG` *`option`* [*`option`* `...`]

Options

The following are valid options for the DEBUG directive.

The default for this option is NOIC86.

IC86
NOIC86

These options enable/disable compatibility with the Intel iC86 compiler. When enabled, the IC86 option restricts the debug information output and folds all symbols to uppercase to match the output of the Intel compiler.

TYPES
NOTYPES

These options enable/disable the inclusion of type records in the output OMF86 file.

PUBLICS
NOPUBLICS

These options enable/disable the inclusion of public symbol records in the output OMF86 file.

SYMBOLS
NOSYMBOLS

These options enable/disable the inclusion of local symbol records in the output OMF86 file.

LINES
NOLINES

These options enable/disable the inclusion of line number records in the output OMF86 file.

This is the default for the DEBUG directive.

ALL

This option enables all debug information in the output OMF86 file and is the same as specifying the TYPES, PUBLICS,

	SYMBOLS and LINES options.
NONE	This option disables all debug information in the output OMF86 file and is the same as specifying the NOTYPES, NOPUBLICS, NOSYMBOLS and NOLINES options.
EXTENSIONS **NOEXTENSIONS**	This option enables or disables the Paradigm OMF86 extensions, which include extended enumerations and C++ support. The default for these options is NOEXT, as extensions may not be compatible with third-party debuggers.
CLASSES **NOCLASSES**	This option enables or disables the output of C++ class type information in the OMF86 output file.
ENUMS **NOENUMS**	This option enables or disables the output of extended enumeration debug information in the OMF86 output file.
BIGTYPES **NOBIGTYPES**	This option enables or disables the output of extended types in the OMF86 output file. Only enable this option if your debugger supports 64K type records.
MEMBER-FUNCTION **NOMEMBER-FUNCTION**	This option enables or disables the output of C++ member function in the OMF86 output file.
DESTRUCTORS **NODESTRUCTORS**	This option enables or disables the output of C++ destructors in the OMF86 output file. Some tools may not be able to handle C++ destructor syntax so enable this option only if it is supported by your tools.
OPERATORS	This option enables or disables the output of C++ operators in the OMF86 output file.

NOOPERATORS	Some tools may not be able to handle C++ operator syntax so enable this option only if it is supported by your tools.
CLASSTEMPLATES **NOCLASS-TEMPLATES**	This option enables or disables the output of C++ templates in the OMF86 output file.
SPACES **NOSPACES**	This option enables or disables the removal of spaces from symbols.
PARAMETERS **NOPARAMETERS**	This option enables or disables the inclusion of function parameters in C++ function names.
SPECIALS **NOSPECIALS**	This option enables or disables the output of special C++ characters in names.
ALLEXTENSIONS **NOEXTENSIONS**	This option enables all C++ extensions, except BIGTYPES or disables all extensions.

Because of limited support for C++ types and symbols, C++ developers may wish to enable additional C++ OMF output.

Command line options

Each of the DEBUG arguments can be specified using the Paradigm LOCATE command line options, as shown below.

```
-Od      NONE
-Od-     ALL
-Oe      EXTENSIONS
-Oe-     NOEXT
-Oi      IC86
-Oi-     NOIC86
-Ol      LINES
-Ol-     NOLINES
-Op      PUBLICS
-Op-     NOPUBLICS
-Ot      TYPES
-Ot-     NOTYPES
-Ox      SYMBOLS
```

```
-Ox-     NOSYMBOLS
-Oea[-]  Enable/disable all C++ extensions
-Oec[-]  Enable/disable C++ class translation
-Oed[-]  Enable/disable C++ destructor support
-Oee[-]  Enable/disable enumeration extensions
-Oem[-]  Enable C++ member function extensions
-Oeo[-]  Enable C++ operator extensions
-Oep[-]  Enable C++ parameter extensions
-Oes[-]  Enable space removal from C++ names
-Oet[-]  Enable/disable OMF large types
-Oex[-]  Enable/disable C++ template support
-Oez[-]  Enable C++ special symbol extensions
```

Notes This directive only affects the output of Paradigm LOCATE when the ABSFILE OMF86 directive or **-Aomf** command line option is in effect.

Examples

```
debug  notypes nosymbols    \
       nopublics nolines

debug  none          // Same as above
```

DISPLAY

Description

The DISPLAY directive controls the level of diagnostic information output emitted by Paradigm LOCATE during the processing of input and output files.

Paradigm LOCATE can display the names of each output file as it is being written, display compression statistics, or track module names as they are processed to indicate the progress towards completion.

Syntax

`DISPLAY` *`option`* [*`option`* `...`]

Options

The `option` argument can be selected from one of the following options

NONE	Disables all display diagnostics.
FILES	Displays the filenames of the output files as they are processed.
MODULES	Displays the modules names found in the input files as they are processed.
COMPRESSION	Enables the display of compression statistics for compressed classes.
ALL	Enables the display of all Paradigm LOCATE diagnostics.

Command line options

The DISPLAY directive options can also be specified from the command line as follows

```
-d0             NONE
-d1             FILES
-d2             MODULES
-d3             COMPRESSION
-d4             ALL
```

Examples

```
display  files compression
display  all
display  none
```

DUPLICATE

Description

The DUPLICATE directive is used to make a copy of a class. Typically, the copy of the class is located in the EPROM address space to be used to initialize RAM by the startup code.

Syntax

`DUPLICATE` *`src_class`* *`dest_class`*

Options

DUPLICATE makes a copy of the class *src_class* and appends it to the class *dest_class*, creating the *dest_class* class if it does not already exist.

Command line options

None

Notes

If the duplicated class already exists, the newly made copy will be concatenated to the existing class; otherwise, the new class is simply created. The segments in the duplicated class keep the same segment names and offsets, but pick up the name of the new class. The same offsets are kept to permit multiple classes to be concatenated together into a single duplicated class while preserving the address relationships between the classes.

This is the method used to make copies of initialized data for placement in EPROM. Since the first segment in the duplicated class has been defined in the startup code and has a physical address, and the length of the original class is known, it is a simple matter for the compiler startup code to copy the class from EPROM to RAM.

Example

```
dup    DATA ROMDATA       // Copy class DATA

class  DATA = 0x0040      // DATA at 00400H
class  CODE = 0xfc00      // CODE at FC000H

order  CODE ROMDATA       // ROMDATA after CODE
```

Description

The HEXFILE directive is used to create hex and binary files suitable for download to an EPROM programmer. You can use as many HEXFILE directives as desired in a configuration file to create any number of different output files.

If you choose to create multiple output files in a single pass of Paradigm LOCATE, be sure to use the FILENAME option to name the output file for each HEXFILE directive so that Paradigm LOCATE will not overwrite any of the files.

Syntax

```
HEXFILE  [ INTEL80|INTEL86|BINARY|TEKHEX ] \
         [ OFFSET=addr]                    \
         [ SIZE=size ]                     \
         [ SPLIT=split ]                   \
         [ FILL=fill ]                     \
         [ LENGTH=len ]                    \
         [ TRUNCATE ]                      \
         [ FILENAME=file ]
```

Options

The following is a description of the HEXFILE options.

INTEL80
INTEL86
INTEL386
BINARY
TEKHEX

These mutually exclusive arguments select one of the following output file types. The number in parentheses indicates the maximum size of the address space supported by each file type.

INTEL80	Intel hex (64KB)
INTEL86	Intel extended hex (1MB)
INTEL386	Intel 386 extended hex (1MB or 16MB)
BINARY	Binary (1MB)
TEKHEX	Tektronix hex (64KB)

INTEL86 is the default output file type for the HEXFILE directive.

If left unspecified, the address for the OFFSET argument defaults to 0.

OFFSET

The OFFSET option is used to select a subset of the 1MB address space. The address defined in addr is used as the base for any subsequent file operations.

For example, to burn a 32KB EPROM using the Intel hex format that starts at address F8000H, the offset field should be set to F8000H, which makes offset 0000H within the EPROM correspond to offset F8000H of the address space.

If you are creating multiple EPROM images using the SPLIT option, each image will be the selected size.

SIZE

The SIZE option is used to set an upper limit on the image size (in KB), up to the upper limit imposed by the output file type. The size field can be any value from 1 (a 1KB EPROM image) up to 1024 (a full 1MB EPROM image).

The default image size is the maximum size of the selected output file type, except for binary files which default to 32K bytes.

Intel extended hex files cannot be split due to the presence of segment records.

SPLIT

The SPLIT option is used to extract a set of 1 to 4 EPROM files from the specified region, where each file corresponds to a memory bank. Splitting the EPROM image is normally required when working with 16- and 32-bit buses implemented using 8-bit wide EPROMs. The split field can take on the values 1, 2 or 4 with 1 being the default value.

This option used to be called PAD in previous versions of Paradigm LOCATE.

FILL

The FILL option is used to inform Paradigm LOCATE of the value of the background fill character for use in binary output. The fill character defines the background pattern for binary files; all other file types require that the EPROM programmer be used to set the fill prior to downloading the file. If not specified, the fill character defaults to 0FFH.

LENGTH

The LENGTH options lets you change the record length for hex output files. The default hex record contains a maximum of 16 bytes per record. Using this option, you can change the record length from 8 to 64 bytes in size.

TRUNCATE This option is used to create binary files having only the data contained in the load module. When this option is not in effect, the size of the binary output files will be determined by the SIZE option. When TRUNCATE is used, the file size will be the minimum of the SIZE option and the offset of the last data written into the file.

FILENAME The FILENAME option sets the output filename for an EPROM image. If left unspecified, the output filename defaults to the same filename as the input file.

> Use slashes (/) instead of backslashes (\ ) for path name separators. For example,
> `file = c:/output/test.hex`

Command line options

The Paradigm LOCATE command line can be used to set the options for a single EPROM image using the following switches:

`-Hb`	`BINARY`
`-Hdsize`	`SIZE=`*`size`*
`-He`	`INTEL86`
`-Hf`*`fill`*	`FILL=`*`fill`*
`-Hi`	`INTEL80`
`-Hl`*`len`*	`LENGTH=`*`len`*
`-Hn`*`file`*	`FILENAME=`*`file`*
`-Ho`*`addr`*	`OFFSET=`*`addr`*
`-Hs`*`split`*	`SPLIT=`*`split`*
`-Ht`	`TEKHEX`

-H options work independent of the configuration file HEXFILE directive. If you have a HEXFILE directive(s) and -H options in one single pass of Paradigm LOCATE, LOCATE will first create all EPROM output based upon the HEXFILE directive, then create an additional EPROM output based solely upon -H command options.

Notes

The following are the file extensions used by the HEXFILE directive. File extensions are determined by the file type and cannot be changed.

File type/Split	1	2	4
INTEL86	.HEX	---	---
INTEL386	.HEX	---	---
INTEL80	.HEX	.HX?	.HX?
BINARY	.BIN	.BN?	.BN?
TEKHEX	.TEK	.TK?	.TK?

Examples

```
hexfile intel86
```

This example creates an Intel extended output file containing all classes identified in OUTPUT directives.

```
hexfile intel80 offset=0xe0000 file=no1
hexfile intel80 offset=0xf0000 file=no2
```

This example is for an 8-bit system having a pair of 64KB EPROMs in the upper 128KB of the address space. Because the Intel hex file format can hold at most 64KB of code/data, two HEXFILE directives are used to create separate EPROM images. The OFFSET option is used with each HEXFILE directive to specify which 64KB of the address space we wish to be extracted and placed in the output file.

```
hexfile intel80 offset=0xf0000 size=32 file=no1
hexfile intel80 offset=0xf8000 size=32 file=no2
```

This example is similar to the preceding example except that only 64KB of address space is available and the SIZE option is used to limit each output file to the 32KB regions of the address space occupied by each EPROM.

```
hexfile intel80 offset=0xc0000 split=2 file=no1
hexfile intel80 offset=0xe0000 split=2 file=no2
```

This final example is for a 16-bit system having a total of 256KB of EPROM divided into two 128KB banks. In this case the output files would be NO1.HX0, NO1.HX1, NO2.HX0, and NO2.HX1, containing the even and odd addresses for each pair of EPROMs.

Description

The INITCODE directive is used to automatically generate reset vectors, stack initialization and peripheral register initialization code. The INITCODE directive accepts a list of peripheral register assignments that depend on the microprocessor type. This permits chip select, DRAM refresh and wait state initialization code to be handled independently of the application and startup code. This makes it a simple task to ensure the physical ROM and RAM in the system are addressable before the application is handed control of the target microprocessor.

If stack or I/O port initialization code is created using this directive, Paradigm LOCATE will automatically change the entry point to ensure that the initialization code, if present, is executed in the following order: reset code, stack initialization, peripheral register initialization code, and the startup code.

Syntax

```
INITCODE   [ RESET | NORESET ]          \
           [ STACK | NOSTACK ]          \
           [ ioport = data ]            \
           [ OUTBYTE addr16 = data ]    \
           [ OUTWORD addr16 = data ]    \
           [ INBYTE addr16 ]            \
           [ INWORD addr16 ]
           [ filename=file.ext CLASS = class_name]
```

Options

The INITCODE directive supports the following options:

RESET
NORESET

The RESET option enables the generation of a far jump instruction at address FFFF0H to the application entry point. The option NORESET disables the creation of the far jump.

STACK
NOSTACK

The STACK option generates code to initialize the SS:SP registers with the default stack (the segment having the stack attribute). If enabled, the stack initialization code will be placed in the segment **??STACKINIT** in class **??LOCATE**.

ioport The *ioport* option accepts processor peripheral registers to be initialized from the configuration file. The order of the I/O or special function register operations is the order of the port arguments in the configuration file INITCODE directive. Any port initialization code created using the INITCODE directive will be placed in the segment **??CPUINIT** in the class **??LOCATE**.

See appendix D on page 163 for a list of supported microprocessors and peripheral registers.

OUTWORD
OUTBYTE
INWORD
INBYTE

General purpose I/O can also be performed using the generic forms for input and output. Note that the input functions discard the input value and are used only for any side effects.

INITCODE *filename=file.ext* CLASS = *class_name* assigns the contents of the binary file *file.ext* to class *class_name* and places the code in the startup code execution list by jumping to the start and appending a far jump to the next code block in the startup code sequence at the end of the file.

Command line options

The following INITCODE arguments can be specified on the command line:

```
-b          RESET
-b-       NORESET
-s          STACK
-s-       NOSTACK
```

Notes

Peripherals in the Intel 80C186-family are accessed using word-aligned byte writes. This allows updating a 16-bit chip select register with a single external bus cycle, even when 8-bit processors are used.

Examples

```
cputype   i80186
initcode  reset          \
          umcs = 0xf038  \  // UMCS value
          lmcs = 0x0ff8     // LMCS value

initcode  outbyte 0xfffe = 0x11
```

LISTFILE

Description

The LISTFILE directive is used to create listing files containing information such as a segment map, lists of public and local symbols and source line numbers. There is no limit on the number of LISTFILE directives used in a configuration file, permitting multiple output files with different reports to be created in a single pass.

If you choose to create multiple output files in a single pass of Paradigm LOCATE, be sure to use the FILENAME option to name the output file of each LISTFILE directive so that Paradigm LOCATE will not overwrite any of the files.

Paradigm LOCATE can only output listings for information to which it has access. If there is no debugging information in the input file, Paradigm LOCATE will be restricted to creating the segment and region maps.

Syntax

```
LISTFILE[ SEGMENTS ]                          \
        [ PUBLICS [(BY ADDRESS|BY NAME ) ] \
        [ COLUMNS=(1 | 2) ]                   \
        [ WIDTH=(80 | 132) ]                  \
        [ SYMBOLS ]                           \
        [ LINES ]                             \
        [ REGIONS ]                           \
        [ CHECKSUMS ]                         \
        [ FILENAME=file ]
```

Options

The following options control the different fields in the Paradigm LOCATE map file:

SEGMENTS The SEGMENTS option is used to create an absolute segment map showing the starting address, ending address and length for each segment in the application.

REGIONS The REGIONS option is used to include a copy of the memory address space assignments specified in the MAP directives and their usage.

CHECKSUMS The CHECKSUMS option is used to include the

details of any checksums or CRCs used by the application, including the starting addr, ending addr and checksum value.

The default for COLUMNS is 1 and the default for WIDTH is 80.

PUBLICS
COLUMNS
WIDTH

The three PUBLICS options are used to control the output of public symbols in the Paradigm LOCATE map file. Used alone, PUBLICS will output the public symbol table sorted first by name and then by address. You may also qualify the output to get one or the other by using the PUBLICS BY NAME or PUBLICS BY ADDRESS arguments.

You can use the COLUMNS and WIDTH options to adjust the number of symbol columns (1 or 2) or the output width (80 or 132 columns) to create an optimally-sized public symbol table.

Be careful - large applications can create very large local symbol list files.

SYMBOLS

The SYMBOLS option controls whether the extended debugging information, such as local symbols, appears in the output file organized by source module.

LINES

The LINES option controls whether or not line number records appear in the output file organized by source module.

FILENAME

This option permits you to change the name of the Paradigm LOCATE listing file to *file*. The default filename is the same as the input file but with the .LOC extension.

> Use slashes (/) instead of backslashes (\ ) for path name separators. For example,
> `file = c:/output/test.loc`

Command line options

The following Paradigm LOCATE command line switches can be used to select the options for a single LISTFILE directive:

```
-Lc             COLUMNS=2
-Ld             CHECKSUMS
-Ll             LINES
-Lnfile         FILENAME=file
-Lp             PUBLICS
-Lr             REGIONS
-Ls             SEGMENTS
-Lw             WIDTH=132
-Lx             SYMBOLS
```

-L options work independent of the configuration file LISTFILE directive. If you have a LISTFILE directive(s) and -L options in one single pass of Paradigm LOCATE, LOCATE will first create all listing output based upon the LISTFILE directive, then create an additional listing output based solely upon -L command options.

Examples

```
listfile  segments file=test.loc
listfile  publics lines symbols segments
```

Description

The MAP directive is used to assign an access attribute to a region of the memory address space. These attributes are then used by Paradigm LOCATE to verify that reserved regions of the memory address space are vacant. Paradigm LOCATE will report the use of reserved regions, or a segment spanning regions with different attributes. Warnings will also be generated if Paradigm LOCATE detects segments mapped in read-only regions are not being output, or segments mapped in non-read-only regions are being output.

Please note that the MAP directive does not assign physical addresses to segments. The purpose of the MAP directive is to describe the target address space partitions so that Paradigm LOCATE can check for overlaps and errors.

Syntax

`MAP [ name ] addr TO addr AS memtype`

Options

The following fields must be defined in each MAP directive.

name An optional name to be associated with the region.

addr ***addr*** The first argument defines the start of the region of the memory address space to be mapped while the second argument defines the end of the region, where the first address must be less than or equal to the second.

memtype The *memtype* field is used to assign one of the following access attributes to the region.

`RDONLY`	Read only address space
`RDWR`	Read/write address space
`RESERVED`	No access
`MMIO`	Memory-mapped I/O
`IRAM`	Internal RAM
`SFR`	Special function registers

Command line options

None

Examples

```
map  my_data  0x00000 to 0x0ffff as rdwr
map  0x10000 to 0xEFFFF as reserved
map  0xF0000 to 0xFFFFF as rdonly
```

ORDER

Description

The ORDER directive is used to concatenate one or more classes relative to the anchoring class. The ORDER directive is important since it allows unrelated classes to be grouped together in the memory address space, independent of the lengths of the individual classes.

Syntax

```
ORDER  anchor_class  class_list
```

Options

The first argument *anchor_class* is the anchor class and must appear in a CLASS directive or in a previous ORDER directive. The classes defined in the argument *class_list* are then located contiguous to the anchor class, subject to the class alignment requirements.

Command line options

None

Examples

```
order  DATA BSS STACK  // RAM classes

order  DATA BSS        // Same as above
order  BSS STACK

order  DATA    \       // Still the same
       BSS     \
       STACK
```

Description

The OUTPUT directive is used to specify the classes containing code or data destined for any of the output files created with the ABSFILE or HEXFILE directives.

Syntax

```
OUTPUT  class_list
```

Options

The argument *class_list* is a list of one or more class names that are to be placed in the output file.

Classes containing program code and constant data must be named in an OUTPUT directive to be available when the system is powered up and initialized. Other classes, such as those containing uninitialized data or the program stack, require only to be assigned a physical address. While these classes are assigned a position within the memory address space, they do not need to appear in the output file since they contain only uninitialized data.

Warnings will be generated if Paradigm LOCATE detects that segments mapped in read-only regions are not in output or segments mapped in non-read-only regions are in output.

Command line options

None

Examples

```
output  CODE ROMDATA  // One style

output  CODE          // Another style
output  ROMDATA
```

SEGMENT

Description

The SEGMENT directive is used to assign a physical address to a segment, independently of the segment's membership in a class. While supported in Paradigm LOCATE, it is strongly recommended that segments be placed in unique classes to place them anywhere in the address space.

Syntax

```
SEGMENT  segname=addr16
```

Options

The segment *segname* is assigned the 16-bit physical segment specified by the *addr16* argument.

SEGMENT directives are always processed before CLASS directives to allow the removal of the segment from the class before physical addresses are assigned to the class.

There is a restriction on the use of the SEGMENT directive in that it cannot be used to set the address of any segment that is a member of a group. Segments within a group have segment fixups relative to the group base and all offsets are from the group base, not the start of the segment.

Command line options

None

Examples

```
segment  MY_CODE=0xfc00
segment  TEST_TEXT=0x0800
```

WARNINGS

Description

The WARNINGS directive is used to enable or disable the warning diagnostics output by Paradigm LOCATE. Either individual warnings or all warnings can be enabled or disabled.

Syntax

```
WARNINGS   ALL              \
           NONE             \
           EXITCODE=n       \
           warn_list
```

Options

The options ALL or NONE enable or disable all warnings.

The EXITCODE option can be used to have Paradigm LOCATE return a non-zero exit code should any warnings be detected during the processing of the input files.

The option warn_list is one or more warning diagnostics identifiers, prefixed with a '+' to enable the warning or a '-' to prevent the warning from being displayed. A list of warnings organized by number can be found in Appendix A of this manual.

The default state for all warning diagnostics is enabled.

The WARNINGS directive is useful to eliminate certain warnings that occur each time Paradigm LOCATE is used, such as the register variable warning for OMF86 output. To disable a warning permanently, you should add the appropriate command line version of this directive to your **LOCATE.OPT** file.

Command line options

The following command line switches can also be used to enable or disable warning diagnostics:

```
-w+Wid      WARNINGS +Wid
-w-Wid      WARNINGS -Wid
-w+         WARNINGS ALL
-w-         WARNINGS NONE

-W-         WARNINGS EXITCODE=0
-W          WARNINGS EXITCODE=1
```

Examples

```
warnings  -w1001 -w1002    \
          +w1004
warnings  exitcode=1
```

C H A P T E R

6

Command line options

In addition to the configuration file directives described in the previous section, Paradigm LOCATE can process options from the command line or a special file that Paradigm LOCATE searches for each time it is run. These options enable the Paradigm LOCATE user to define the default behavior of Paradigm LOCATE and provide a convenient means to override the default when circumstances dictate a different response. Whether defined on the command line or in an option file, the syntax used for the command line options is the same.

Command line options

When defined on the command line, all Paradigm LOCATE options are preceded by the hyphen ('Ä') character and are separated from the Paradigm LOCATE program name, any other command line options, and the application filename by one or more spaces or tabs.

```
locate [ option [ option ... ] ] filename
```

where the *filename* defaults to the extension .ROM. Paradigm LOCATE will then look for the files *filename*.MAP and *filename*.CFG, unless overridden by command line options.

The following are some typical examples of Paradigm LOCATE command line options:

```
locate -b myfile
locate -Aomf -Anherfile.omf herfile
```

LOCATE.OPT file

In addition to the options specified on the command line, additional options can be placed in the **LOCATE.OPT**; option file. **LOCATE.OPT** options can be listed on the same line separated by spaces or tabs or can be placed on multiple lines as shown below.

```
-Xoomf -Xlmp2
-Aomf
```

When you run Paradigm LOCATE, it looks for **LOCATE.OPT** in the current directory. If it is not found and you are running DOS 3.3 or higher, the directory containing Paradigm LOCATE will be searched for this file.

Option priorities

We have seen that Paradigm LOCATE can receive option input from three different sources; the command line, the **LOCATE.OPT** file, and the configuration file. Should conflicting options be specified, the processing order (from lowest to highest priority) of options is:

- **LOCATE.OPT** options
- configuration file directives
- command line options

This processing order permits options defined in either the configuration file or on the DOS command line to override the default options in the **LOCATE.OPT** file, while command line options can also be used to override any options specified in the configuration file.

Summary of options

Table 6.1 is a summary of the command line options accepted by Paradigm LOCATE. Each of the options is described in further detail later in this section, where the different options are organized into related groups.

Table 6.1
Command line summary

Option	Page	Function
-A*pdxx*	92	Select AXE86 absolute file output
-Ad	92	Disable absolute output file
-An*file*	92	Supply a filename for the absolute output file
-A*omf*	92	Select OMF86 absolute file output
-b	82	Enable reset vector generation
-b-	83	Disable reset vector generation
-c*file*	87	Specify a different configuration file name
-D*macro*	82	Define macro
-D*macro=text*	82	Define macro to text
-d0	83	Disable processing diagnostics
-d1	83	Enable filename processing diagnostics
-d2	83	Enable filename and module processing diagnostics
-d3	84	Enable compression diagnostics
-d4	84	Enable all processing diagnostics
-Ee	84	Enable the error/warning log
-En*file*	84	Supply a filename for the error/warning log
-Hb	88	Select a binary EPROM output file
-Hd*size*	88	Specify the EPROM size in KB
-He	88	Select Intel extended hex EPROM output
-Hf*fill*	88	Specify the EPROM fill character
-Hi	88	Select Intel hex EPROM output
-Hl*len*	89	Select hex record length
-Hn*file*	89	Supply a filename for the EPROM output file
-Ho*addr*	89	Specify the EPROM file offset
-Hs*split*	89	Specify the EPROM split size
-Ht	89	Select Tektronix hex EPROM output
-Lc	90	Set public symbol display columns to 2
-Ld	90	Write checksum statistics to listing file
-Ll	90	Write line numbers to listing file
-Ln*file*	90	Supply a filename for listing file
-Lp	91	Write public symbols to listing file
-Lr	91	Write the region map to listing file
-Ls	91	Write the segment map to listing file
-Lw	91	Set public symbol output width to 132 columns
-Lx	91	Write extended debug information to listing file
-Od[-]	85	Enable/disable all OMF86 debug information
-Oe[-]	85	Enable/disable Paradigm OMF86 extensions

-Ol[-]	86	Enable/disable output line number records
-Op[-]	86	Enable/disable public records in OMF86 output
-Ot[-]	87	Enable/disable type records in OMF86 output
-Ox[-]	87	Enable/disable symbol records in OMF86 output
-q	83	Disable sign on displays
-s	83	Enable stack initialization code
-s-	83	Disable stack initialization code
-W	84	Enable a non-zero exit code on warnings
-w+	85	Enable the display of all warnings
-w-	85	Disable the display of all warnings
-w-*Wxxxx*	85	Disable the display of warning Wxxxx
-w+*Wxxxx*	85	Enable the display of warning Wxxxx
-Xa*ext*	93	Set the default AXE86 output file extension
-Xc*ext*	93	Set default configuration file extension
-Xl*ext*	93	Set the default listing file extension
-Xm*ext*	93	Set the default linker map file extension
-Xo*ext*	93	Set the default OMF86 output file extension

Defining macros

Macros for the Paradigm LOCATE configuration file can be defined on the command line with the **-D** command line option.

-D*name* Defines the macro identifier *name* and sets its value to 1.

-D*name=text* Defines the macro identifier *name* and sets its value to *text*.

Initialization

The following options permit Paradigm LOCATE to automatically generate the reset vector and stack initialization code:

-b Enables the automatic creation of a reset vector pointing to the program entry point and places the code at the absolute address FFFF0H.

Directive: INITCODE RESET

-b- Disables any reset vector code generation (the default).

Directive: INITCODE NORESET

-s Enables the automatic creation of initialization code for the SS:SP register pair and places it in the class **??LOCATE**.

Directive: INITCODE STACK

-s- Disables any stack initialization code (the default).

Directive: INITCODE NOSTACK

Diagnostics

The following set of options control the display of diagnostic messages. Paradigm LOCATE gives you complete control over the display of output diagnostics and log files, plus the ability to customize the display of individual warning messages.

Startup display

These options control the display of the Paradigm LOCATE copyright and version information when Paradigm LOCATE is first started.

-q Disables the output of Paradigm LOCATE copyright and version displays.

Processing diagnostics

Processing diagnostics enable Paradigm LOCATE to keep you informed of which files and modules are being processed and where errors and warnings are being generated.

-d0 Disables the output of all processing diagnostics (the default).

Directive: DISPLAY NONE

-d1 Enables the display of the filename of each file as it is processed by Paradigm LOCATE.

Directive: DISPLAY FILES

-d2 Enables the display of the filename of each file as it is processed by Paradigm LOCATE, along with the

module names from the input files. This mode is especially useful to help identify which of the input modules is generating errors or warnings.

Directive: DISPLAY MODULES

-d3 Enables the display of compression diagnostics. Use this display mode to see how much Paradigm LOCATE is compressing your classes.

Directive: DISPLAY COMPRESSION

-d4 Enables the display of all diagnostics.

Directive: DISPLAY ALL

Error/warning log

Paradigm LOCATE can keep a log file containing all errors, warnings and output diagnostics. These options allow you to enable, disable and name the error log managed by Paradigm LOCATE.

Use -Ee- to disable the error log.

-Ee Enables the creation of an error/warning log file. Unless overridden with the **-En** option, the log will have the same filename as the input file with the .ERR extension.

-En*file* Specifies a filename to be used for the error/warning log and enables logging diagnostic output to the file. If no filename is specified in the file field, Paradigm LOCATE will use the default filename for log files.

Exit code control

By default, Paradigm LOCATE returns a zero exit code if processing is successfully completed without any errors. If it is desirable to have Paradigm LOCATE return a non-zero exit code when warnings have been issued, such as to stop a build by a MAKE utility, the **-W** command line option can be used.

-W Enables Paradigm LOCATE to return a non-zero exit code when warnings have been issued.

-W- Disables Paradigm LOCATE from returning a non-

zero exit code when warnings have been issued.

Warning diagnostic control

The warning control options permit individual warnings to be enabled or disabled, making it easy to filter out any warnings which are harmless but distracting.

-w- Disables the display of all warning diagnostics.

Directive: WARNINGS NONE

-w+ Enables the display of all warning diagnostics (the default).

Directive: WARNINGS ALL

-w-W*xxxx* Disables the display of warning W*xxxx*.

Directive: WARNINGS -Wxxxx

-w+W*xxxx* Enables the display of warning W*xxxx*.

Directive: WARNINGS +Wxxxx

OMF86 debug control

The following group of command line options control how debug information is treated as the input files are processed into OMF86 output files. By eliminating unnecessary debugging information, the output file size is reduced and processing speeded up.

-Od Places all debugging records in the OMF86 output file (the default).

Directive: DEBUG ALL

-Od- Disables all debugging records from appearing in the OMF86 output file.

Directive: DEBUG NONE

-Oe Enables the use of the Paradigm OMF86 debug extensions. See the description on page 59 for the list of supported OMF86 extensions.

-Oe- Disables the use of Paradigm OMF86 debug extensions (the default).

Directive: DEBUG NOEXT

-Oi Enables the output of an Intel iC86-compatible OMF86 file. Intel iC86 supports only one scope per function, folds all symbols to uppercase and does not use leading underscores on public symbols. With this option enabled, Paradigm LOCATE will output an OMF86 file that closely matches the output from the Intel compiler.

Directive: DEBUG IC86

-Oi- Disables the output Intel iC86-compatible OMF86 (the default).

Directive: DEBUG NOIC86

-Ol Enables the output of line numbers in the OMF86 output file.

Directive: DEBUG LINES

-Ol- Disables the output line numbers in the OMF86 output file. Use this option to strip out line numbers if they are not needed by your debugger or in-circuit emulator.

Directive: DEBUG NOLINES

-Op Enables the output of public symbols in the OMF86 output file.

Directive: DEBUG PUBLICS

-Op- Disables the output of public symbols in the OMF86 output file. Use this option to strip out public symbols if they are not needed by your debugger or in-circuit emulator.

Directive: DEBUG NOPUBLICS

-Ot Enables type information in OMF86 output.

Directive: DEBUG TYPES

-Ot- Disables the output of type information in the OMF86 output file. Use of this option to eliminate type information if not needed by your debugger or in-circuit emulator.

Directive: DEBUG NOTYPES

-Ox Enables the output of extended debug information (local symbols and scopes) in the OMF86 output file.

Directive: DEBUG SYMBOLS

-Ox- Disables extended debug information in OMF86 output.

Directive: DEBUG NOSYMBOLS

File management

The remaining options have to do with managing the files created and used by Paradigm LOCATE.

Configuration files

This option permits any file to be used in place of the default configuration file for Paradigm LOCATE.

-c*file* Use the filename *file* as the Paradigm LOCATE configuration file. If not specified in this option, Paradigm LOCATE will use the filename from the load module with the .CFG extension (unless changed with the **-Xc** option).

EPROM files

This group of options control the creation of files suitable for download to an EPROM programmer. The output file(s) will have the same filename as the input file with the extension determined by the file type and number of splits.

These options can process at most one EPROM image from the command line. Using the configuration file HEXFILE directive, as many EPROM images as desired can be created in a single pass of Paradigm LOCATE.

-H options work independent of the configuration file HEXFILE directive. If you have a HEXFILE directive(s) and -H options in one single pass of Paradigm LOCATE, LOCATE will first create all EPROM output based upon the HEXFILE directive, then create an additional EPROM output based solely upon -H command options.

-Hb Selects the binary EPROM format for the output file. This file format can hold up to 1MB of data.

Directive: HEXFILE BINARY

-Hd*size* Allows the EPROM size to be selected. The unit of measurement for the *size* argument is in KB and can be value from 1 (1KB EPROM image) to 1024 (a 1MB EPROM image).

Directive: HEXFILE SIZE=*size*

-He Selects the Intel extended hex EPROM format for the output file. This file format can hold up to 1MB of data.

Directive: HEXFILE INTEL86

-Hf*fill* Permits the specification of the fill character for the unused locations in the EPROM image. Only binary output files will contain the fill character; all other formats use it only in checksum/CRC calculations and it must be set by the EPROM programmer before loading the EPROM image. The default fill character is 0xFF.

Directive: HEXFILE FILL=*fill*

-Hi Selects the Intel hex EPROM format for the output file. This format can hold up to 64KB of data.

Directive: HEXFILE INTEL80

-Hl*len* This options allows the size of the hex file data records to be adjusted between 8 and 64 bytes per record.

Directive: HEXFILE LENGTH=*len*

-Hn*file* Specifies a filename to be used for the output file(s). Note that the file extension is determined by output file type and split (see the HEXFILE directive for a table of file extensions). If no filename is specified in the *file* field, Paradigm LOCATE will use the default filename.

Directive: HEXFILE FILENAME=*file*

-Ho*addr* Allows the specification of an address space offset to permit Intel hex, binary and Tektronix hex files to select the subset of the 1MB address to be included in the output file. The argument *addr* is a 20-bit physical address and defaults to zero if not specified.

Directive: HEXFILE OFFSET=*addr*

-Hs*split* Specifies the EPROM split count (1, 2 or 4) in the *split* argument. Splitting Intel extended hex files is not allowed as they contain segment information. The default split is 1.

Directive: HEXFILE SPLIT=*split*

-Ht Selects the Tektronix hex EPROM format for the output file. This format can hold up to 64KB of data.

Directive: HEXFILE TEKHEX

Listing files

This group of options control the creation of a listing file containing design documentation using the target system addresses. The output

file will have the same filename as the input file with the .LOC extension (unless changed with the **-Xl** option).

This option can process at most one listing file from the command line. Using the configuration file LISTFILE directive, as many listing files as desired can be created in a single pass of Paradigm LOCATE.

-L options work independent of the configuration file LISTFILE directive. If you have a LISTFILE directive(s) and -L options in one single pass of Paradigm LOCATE, LOCATE will first create all listing output based upon the LISTFILE directive, then create an additional listing output based solely upon -L command options.

-Lc Sets the number of symbol columns for the public symbol tables to use two columns. This option results in a more compact display when many public symbols are part of the application.

Directive: LISTFILE COLUMNS=2

-Lc- Sets the number of symbol columns for the public symbol tables to use a single column.

Directive: LISTFILE COLUMNS=1

-Ld[-] This option enables the output of the checksum map to the listing file. If no CHECKSUM directives are present in the configuration file, no output will be generated.

Directive: LISTFILE CHECKSUMS

-Ll[-] Writes the source module name and line numbers to the listing file. If no line numbers are present in the input file, no output will be generated.

Use the **-Ll-** option to disable the inclusion of line number in the listing file.

Directive: LISTFILE LINES

-Ln*file* Supplies a filename for the listing file. If no filename is specified in the *file* field, Paradigm

LOCATE will use the default filename.

Directive: LISTFILE FILENAME=*file*

-Lp[-] Writes the public symbols sorted by name and by address to the listing file. If no public symbols are present in the input file, no output will be generated.

Directive: LISTFILE PUBLICS

-Lr Writes the memory address space attribute map to the listing file.

Directive: LISTFILE REGIONS

-Ls Writes the absolute segment map to the listing file.

Directive: LISTFILE SEGMENTS

-Lw Sets the width of the output for the public symbol table to 132 columns. Using this option can prevent the clipping of public symbols when the two column format is used.

Directive: LISTFILE WIDTH=132

-Lw- Sets the width of the output for the public symbol table to 80 columns.

Directive: LISTFILE WIDTH=80

-Lx Writes the local symbols and other debugging information to the listing file. If the extended debug information is not available in the input file, no output will be generated.

Directive: LISTFILE SYMBOLS

Absolute files

These options control the type of absolute output file created by Paradigm LOCATE. Unless you plan to use a debugger (like Paradigm DEBUG or the Turbo Debugger) or an in-circuit emulator, there is no need to create an absolute output file with debug

information. These options can be set from the configuration file using the ABSFILE directive.

-Ad Disables the creation of any absolute output file (the default).

Directive: ABSFILE NONE

-An*file* Supplies a filename to be used for the absolute output file. If no filename is specified in the *file* field, Paradigm LOCATE will use the default filename.

Directive: ABSFILE FILENAME=*file*

-Aomf Selects an Intel OMF86 output file. The output file will have the same filename as the input file with the .ABS extension (unless changed with the **-Xo** option). The format and debug information content of the OMF86 file are controlled by the **-D?** options.

Directive: ABSFILE OMF86

-Apd60
-Apd50
-Apd40
-Apd31
-Apd30
-Apd20
-Apd10

Selects the Paradigm AXE86 output file format for a specific version of Paradigm DEBUG. The output file will have the same filename as the input file with the .AXE extension (unless changed with the **-Xa** option).

Directive: ABSFILE AXE86

Filename extensions

Paradigm LOCATE comes with a set of default file extensions for input and output files but you can choose your own if you don't care for the default extensions. While these options can be used on the command line, they are much better suited for inclusion in the **LOCATE.OPT**; file.

None of these options can be set with configuration file directives.

The argument *ext* in the **-X?** options must be three characters or less; otherwise an error will be reported. If no file extension is specified,

the **-X?** switch will restore the default file extension used by Paradigm LOCATE.

-Xa*ext* Sets the default file extension for files using the Paradigm AXE86 format.

Default: .AXE

-Xc*ext* Sets the default file extension for the Paradigm LOCATE configuration file.

Default: .CFG

-Xlext Sets the default file extension used by listing files created with the **-L?** options or the LISTFILE directive.

Default: .LOC

-Xm*ext* Sets the default file extension used to open the segment map produced by the linker.

Default: .MAP

-Xo*ext* Sets the default file extension used for output files in the Intel OMF86 format.

Default: .ABS

C H A P T E R

7

Checksums and CRCs

Adding checksums or CRCs (cyclic redundancy checks) to an application can provide a higher degree of protection against the failure of a device in the field, or the ability to detect an incorrect update of a system employing technology such as flash EPROMs.

Paradigm LOCATE offers a number of checksum and CRC options, each designed to address the needs of applications using embedded PCs, or those that need the greatest degree of fault protection in the target system. The CHECKSUM directive is used to define a region of the target system address space to be included in a checksum or CRC calculation, the background fill to be used by any undefined addresses within the region, and optionally specify the exact position to place the calculated checksum or CRC.

Look for the example CRCDEMO in the EXAMPLES subdirectory for your compiler.

Here we introduce the general concept of a checksum or CRC but we don't go into too much detail as there is nothing better than a working example. For more information and for an actual application employing checksums and CRCs, see the compiler examples available on the Paradigm LOCATE distribution disk.

ROMBIOS checksums

The CHECKSUM directive uses the ROMBIOS option to select the checksum technique used by the IBM PC ROM BIOS for ROM BIOS extensions. This technique uses a simple sum of bytes, carries

ignored, which must sum to zero to be accepted as a legitimate ROM BIOS extension. The PC ROM BIOS scans the ROM BIOS address space looking for the signature bytes, 55H, AAH, followed by the count of 512 byte blocks when performing the extension ROM BIOS scan during the BIOS initialization phase.

If the ROM BIOS finds a valid signature during the expansion ROM BIOS scan, the ROM BIOS will calculate the checksum of the region using the block count field. If the checksum is zero, the ROM BIOS will perform a far call to the ROM BIOS extension entry, which is located immediately following the expansion ROM BIOS block count field.

You can also set the background fill that will be used so the final checksum calculation is correct.

Defining a PC ROM BIOS extension requires that the signature and block size be added to the start of a segment that will be placed on a 2KB boundary. The CHECKSUM directive placed in the Paradigm LOCATE configuration file should look like:

```
CHECKSUM addr1 TO addr2 ROMBIOS
```

where *addr1* and *addr2* define the size of the ROM BIOS extension, minus one, since the default position for the calculated PC BIOS extension checksum is immediately following the end of the region. You can also place the address elsewhere using the ADDRESS option, but the checksum byte must be within the region of the memory address space determined by the signature and block count in order to be recognized as a legitimate ROM BIOS extension.

For example, the CHECKSUM directive for a ROM BIOS extension occupying the region E0000H to 0EFFFFH would be

```
CHECKSUM 0xe0000 TO 0xefffe ROMBIOS
```

If a non-zero fill value is used, the CHECKSUM directive FILL option must be used as it will affect the calculated checksum.

CRC-16 checksums

The CRC-16 checksum is an improvement over the simple sum of bytes used in the expansion ROM BIOS checksum. When an application requires better odds in detecting an error condition, a CRC

check is much more capable of finding not only single errors, but also multiple errors.

You can also specify the background fill if it is not zero.

Defining a CRC-16 checksum is done in an identical fashion to that used in the PC ROM BIOS example:

```
CHECKSUM addr1 TO addr2 CRC16
```

where *addr1* and *addr2* define the size of the address to have the CRC-16 calculated, minus two, since the default position for the calculated CRC-16 is immediately following the end of the region. (We need to leave the last two bytes free to hold the calculated CRC.) If necessary, you can specify a different address to hold the calculated CRC using the ADDRESS option. Unlike the PC ROM BIOS extension, you could store the calculated CRC separately, as shown in the following example:

```
CHECKSUM 0xE0000 TO 0xFFFFF CRC16 ADDRESS=0x80000
FILL=0xff
```

If the checksum is included in the CRC calculation, the result should be zero.

CRC-16 details

The polynomial and initial value used by Paradigm LOCATE to calculate the 16-bit CRC is

Figure 7.1 CRC-16 polynomial and initial value.

```
0xA001U (polynomial)
0x0000U (initial value)
0x0000U (final value)
```

The following C code can be used to calculate the 16-bit CRC in the target system and is taken from the file CRC16.C, available in the EXAMPLES\CRCDEMO directory on the Paradigm LOCATE distribution disk. This is a complete working example which defines a CRC-16 region and verifies that the checksum is correct. For more information on the CRC-16 polynomial and the initial value, refer to the file CHECKSUM.H in the same directory.

Figure 7.2 CRC-16 checksum algorithm.

```
/* Pass thru the buffer and add the new data to the checksum */
wCRC = CRC16_INIT ;
while (dwStart <= dwStop)   {
   /* Build a pointer to the start of the next calculation */
   pByte = MK_FP((UINT)(dwStart >> 4), (UINT)(dwStart & 0xf)) ;
```

```
   /* Compute the size of the buffer */
   wSize = (UINT) min(CRC_BUFSIZE, dwStop - dwStart + 1) ;

   /* Adjust the starting position by the buffer size */
   dwStart += wSize ;

   /* Calculate the CRC on the region */
   while (wSize--)   {
      wIndex = (UINT8) (*pByte++ ^ wCRC) ;
      wCRC >>= 8 ;
      wCRC ^= wCRCTable[wIndex] ;
   }
}

/* Return the computed CRC */
return   wCRC ^ CRC16_FINAL ;
```

CRC-32 checksums

The CRC32 option works identically as the 16-bit CRC option, except that a different polynomial and algorithm is used.

Defining a CRC32 checksum is done in an identical fashion:

```
CHECKSUM addr1 TO addr2 CRC32
```

where *addr1* and *addr2* define the size of the address to have the CRC-32 calculated, minus four, since the default position for the calculated CRC-32 is immediately following the end of the region. (We need to leave the last four bytes free to hold the calculated CRC.) If necessary, you can specify a different address to hold the calculated CRC using the ADDRESS option.

CRC-32 details

The polynomial and initial value used by Paradigm LOCATE to calculate the 32-bit CRC is

Figure 7.3 CRC-32 polynomial and initial value.

```
0xEDB88320UL (polynomial)
0xFFFFFFFFUL (initial value)
0xFFFFFFFFUL (final value)
```

The following C code is used to calculate the 32-bit CRC in the target system and is taken from the file **CRC32.C**, available in the EXAMPLES\CRCDEMO directory on the Paradigm LOCATE distribution disk. This is a complete working example which defines a CRC-32 region and verifies that the checksum is correct. For more

information on the CRC-32 polynomial and the initial value, refer to the file **CHECKSUM.H** in the same directory.

Figure 7.4 CRC-32 checksum algorithm.

```
/* Pass thru the buffer and add the new data to the checksum */
dwCRC = CRC32_INIT ;
while (dwStart <= dwStop)   {
   /* Build a pointer to the start of the next calculation */
   pByte = MK_FP((UINT)(dwStart >> 4), (UINT)(dwStart & 0xf)) ;
   /* Compute the size of the buffer */
   wSize = (UINT) min(0x8000, dwStop - dwStart + 1) ;

   /* Adjust the starting position by the buffer size */
   dwStart += wSize ;

/* Calculate the CRC on the region */
   while (wSize--)   {
      wIndex = (UINT8) (*pByte++ ^ dwCRC) ;
      dwCRC >>= 8 ;
      dwCRC ^= dwCRCTable[wIndex] ;
   }
}
/* Return the computed CRC */
return   dwCRC ^ CRC32_FINAL ;
```

Note that the 32-bit CRC32 result will not be zero if the CRC is included in the CRC calculation.

Tech tips

Here are some useful tips to help you get the most out the Paradigm LOCATE checksum options:

- Use the CHECKSUM FILL option to set the default state for any memory regions that are undefined (and do the same with the debugger before loading the application)
- When debugging, avoid the use of software breakpoints in a checksum region (they will change the checksum calculation)
- Make sure that all classes in the checksummed region are named in OUTPUT directives
- LISTFILE CHECKSUMS option displays the details of any checksums used including the checksum region address range and checksum value.

C H A P T E R

8

Using compression

Paradigm LOCATE offers a compressed data option for applications that require a small EPROM footprint yet have modest amounts of code or initialized data that must be copied from EPROM to RAM at startup. By discussing the various tradeoffs associated with compression, we hope to lend some insight into when it is appropriate to use this advanced feature of Paradigm LOCATE and when it should not be considered.

There are no concrete guidelines when an application should use and when it is best to avoid compressed initialized data. While the impact on the EPROM footprint can be significant, compression, (actually decompression), will cost time as decompression can be from 20 to 50 times slower than straight copying of initialized data from EPROM. Careful consideration of the different options available can make for an optimally designed system if the tradeoffs are well understood.

Likewise, the selection of compiler will also prove to be an important factor in whether compression/decompression will be part of your embedded application. The supported compilers vary in their ability to keep constant data and string literals in the EPROM address space, where the need for copying or decompression can be completely avoided. In extreme cases, it may be preferable to select the compiler on the basis of its ability to control the placement of data, just as one would select the fastest compiler if speed were the dominating factor.

Compression requirements

Adding compressed data to an application requires that Paradigm LOCATE compress the class and output it to an address within the EPROM address space. In the target system, the decompression module must be given the source and destination addresses of the compressed data, and sufficient stack space to perform the decompression.

Check out the COMPRESS example to see the use of compression on FAR DATA class

The interface code to the decompression routine lies in a compiler helper file supplied as part of the Paradigm LOCATE compiler support package. When enabled, this code will pass the default source and destination addresses to the decompression function. If you wish to add you own compressed classes, this code will need to be modified to include support for the additional classes.

Note on decompression stack size.

The final requirement is sufficient stack space for the decompression code to do its work. While no static data is required, the class decompression code requires slightly more than 5KB of stack space during the actual decompression phase. Once completed the stack size can be set to accommodate the run-time needs of the application.

Compiler overview

See the Paradigm BBS for compiler-specific application notes.

This section is devoted to a discussion of the Borland and Microsoft compilers and how they can impact a decision to either use or forego the use of compression. Because of the constantly changing compiler scene, we can only address the latest Borland and Microsoft compilers.

Borland C++

Starting with Borland C++ 4.0, embedded system developers have two means of controlling the placement of string literals and other initialized data. Using the **-dc** command line option, Borland C++ will automatically place string literals in the code segment, a feature long asked for by developers. Controlling the placement of other data is done with the **-z?** command line options, which like the **-dc** option, can be used in source code pragmas to control things on an individual basis.

String literals

Before Borland C++ 4.0, string literals were placed in class DATA and not much could be done about it, without resorting to segregating the data into different source modules and compiling with the **-z?** options. Now with the ability to have the compiler place string literals in the code segment via the **-dc** option, Borland C++ 4.0 users (and later) have a convenient method of getting all string literals out of class DATA and into the code segment without changing a line of source code.

Initialized data

*The **#pragma option** doesn't work correctly in all Borland C++ 3.x releases.*

The **-z?** options, either from the command line or in a **#pragma option**, are used to control the placement of code or data declarations. These options include the ability to specify the segment, class, or group name, with command line options used to cover an entire module, and the **#pragma option** to control data/code placement for individual declarations within a module.

For example, to declare a constant array of integers in the code segment, the following approach is taken:

```
#pragma option -zE_TEXT -zFCODE -zH
const int __far aIntArray[] = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 0
          } ;
```

Other data types are declared in a similar fashion. By using the **-z?** options, in either format, complete control over the location of your code and initialized data is yours.

Microsoft C/C++

The techniques for handling constant data using Microsoft C/C++ are centered on the use of the **__based** keyword, which supplies the ability to control the segment and class for the data.

String literals

Unfortunately, the Microsoft C/C++ compiler does not support placing string literals in the code segment using a pragma or compiler option, but it does have a useful alternative which can result in the same effect. The trick is to use the **__based** keyword to declare the string

literals as initialized character arrays. The array is then used in place of the string literal.

```
const char __based(__segname("_CODE")) aLiteral1[] = "Hello #1" ;
const char __based(__segname("_CODE")) aLiteral2[] = "Hello #2" ;
```

A macro can be defined to help streamline the process, while making the code more portable should a switch to a different compiler be desired.

```
#define  BASED_CODE   __based(__segname("_CODE"))
```

This macro could then be used in the above declarations

```
const char BASED_CODE aLiteral1[] = "Hello #1" ;
const char BASED_CODE aLiteral2[] = "Hello #2" ;
```

Should you later decide to change compilers, the macro can be redefined to be an empty string, effectively eliminating the based code from the application.

Initialized data

Other constant initialized data is declared in a similar fashion to that shown for string literals. For example, if you need to declare an array of string literals, you need to first declare the string literal character arrays and then use each name as initializers in the array declaration.

```
const char far * __based(__segname("_CODE")) aStringArray[] = {
                              aLiteral1,
                              aLiteral2,
                           } ;
```

More information

Only the simplest use of the **__based** keyword has been presented. Based code, data, and pointers are quite powerful and some applications may benefit from the use of the technique in controlling the exact placement of code and/or data. For more information, you may wish to consult the Microsoft C/C++ documentation.

Compression algorithm

The compression algorithm used by Paradigm LOCATE is a variant of the LZW algorithm. This algorithm was chosen over competing

algorithms for its ability to highly compress the most commonly found initialized data types, an ability the other compression algorithm candidates lack.

Most competing solutions use a variant of the run length encoding (RLE) algorithm which compresses repeating sequences of 8- or 16-bit data. While an RLE algorithm works well with segments or classes initialized to a constant value, it fails to deliver acceptable compression on string literals, arrays of data, or lookup tables. Since all classes initialized to a constant value, such as the Borland or Microsoft BSS class, have alternative initializations that are faster and occupy less space, the RLE algorithm typically fails to deliver acceptable performance on the most commonly encountered data types.

C H A P T E R

9

Borland C++ guide

This chapter is a comprehensive overview to using Borland C++ with embedded system applications. Covered in detail are the individual components of the Borland C++ compiler support package, including the startup code, run-time library helper functions, and example applications.

An understanding of the Borland C++ memory models is also very useful.

If you are just getting started with Borland C++, you will need to be familiar with the Borland compiler, Turbo Assembler, TLINK, and MAKE utilities in order to use this chapter. For those who would rather develop using the Integrated Development Environment (IDE), an add-on for using this powerful tool is available. Also covered is a complete description of the standard makefiles shipped with each example in the Borland C++ compiler support package.

Those readers who have worked with Borland C++ in past, should have little trouble working with this compiler in embedded system applications. The Borland C++ compiler brings the best of PC application development tools to the embedded system user, at a price/performance ratio not matched by traditional embedded system development tools.

Startup code

This section covers the files in the BCPP50 root directory.

NOTE: Because the startup module defines the order and alignment of segments, it MUST be linked in first.

The startup code files take an application from the reset vector to the start of your application. Before **main()** can be handed control of the processor, the application stack and segment registers must be initialized, data copied from EPROM to RAM, and the run-time libraries installed and initialized. Even though a lot of work must be accomplished to prepare an application for execution, you rarely will need to modify the startup code. Using the Paradigm LOCATE INITCODE directive and the startup code initializers, applications can add extensions without modifying any of the Paradigm-supplied startup code.

BCPP50.ASM

If you stop and take a peek inside the startup code you will find the following organization. Following the opening comments is a series of **DefSeg** macros that define the default set of segments/classes used by a Borland C++ embedded application.

*Figure 9.1 Segment ordering and alignment with **DefSeg**.*

```
;
; Segment and group declarations.  The order of these
; declarations is used to control the order of segments in the
; .ROM file since the Borland linker copies segments in the
; same order they are encountered in the object files.
;

DefSeg   _TEXT,      para,  public,  CODE,       <>
DefSeg   _INIT_,     para,  public,  INITDATA,   IGROUP
DefSeg   _INITEND_,  byte,  public,  INITDATA,   IGROUP
DefSeg   _EXIT_,     dword, public,  EXITDATA,   IGROUP
DefSeg   _EXITEND_,  byte,  public,  EXITDATA,   IGROUP
DefSeg   _rd,        para,  public,  ROMDATA,    <>
DefSeg   _erd,       para,  public,  ENDROMDATA, <>
```

If you were to take a closer look and examine the definition of **DefSeg** (in the file **STARTUP.INC**), you would discover that these macros don't define any code or data - they simply open and immediately close the segment. Instead of defining code or data, the position of **DefSeg** macros is used by the startup code to set the order and alignment of the default segments and classes required by a Borland C++ application.

Following the segment ordering code is the startup code entry point, which also happens to be the start of the application code. At reset, the startup code must take care of the following chores in order to jump start the application:

1. stack initialization
2. copy the initialized data from EPROM to RAM
3. clear the uninitialized data
4. execute any initializers/constructors present
5. call **main()**

Steps 1 and 5 should be obvious and need no further comment. Steps 2 and 3 handle the initialization of the classes DATA and BSS, which are defined by Borland C++ to contain the initialized and uninitialized data for the application. Since initialized data can only exist in EPROM when power is first applied to the system, step 2 copies any initialized data from its position in EPROM to its position in RAM. The size of the memory block to be copied is determined by measuring the distance from the start of class DATA to the start of class BSS.

Uninitialized data refers to the C/C++ language declaration. The data is actually initialized to zero.

Step 3 fills the class BSS, which contains all the uninitialized static data in your application, with zero bytes. The size of the BSS class is determined by the start of the class BSSEND. We can't use the stack segment as the end of the BSS class because the stack is not part of the group DGROUP in the compact and large memory models.

The work done by step 4 can seem rather complicated but really it is quite simple. Our goal is to automatically initialize a run-time library package without requiring source code modification. Figure 9.2 is the assembly representation of the data structure used in the initializer segment.

Figure 9.2
Initializer structure format

```
InitRec  struc
ctype    db   ?      ; 0=near, 1=far, 0ffh=not used
pri      db   ?      ; 0=highest, 0ffh=lowest
foff     dw   ?      ; Function offset
fseg     dw   ?      ; Function segment
InitRec  ends
```

Every run-time library package or C++ object requiring initialization places an entry into the segment containing an array of these

structures. The startup code then executes each initializer in order of priority before calling **main()**. Creating an entry is even simpler; just use **#pragma startup** to identify the function to be executed by the startup code. Everything else is automatic. If the module is linked in, its initializer will be called and we never have to modify any source code.

Finally, you may be wondering where the peripheral register initializations for chip selects, wait states, and DRAM refresh are located in the startup code. This is handled by the Paradigm LOCATE INITCODE directive so there is no need to modify the startup code to adjust for different target systems. Here is a Paradigm LOCATE configuration file fragment showing sample Intel 80C186EA/XL chip select initialization code.

Figure 9.3 INITCODE reset and chip select register initialization

```
initcode  reset           \     // Reset vector
          umcs = 0xf838   \     // 32KB EPROM
          lmcs = 0x0ff8         // 64KB RAM
```

The startup code not only initializes the stack, but also defines the size of the stack using the macro STKSIZE.

FARDATA.ASM

This module handles the class **FAR_DATA**, if it is present in your application. If you application does not use any far data, then **FARDATA.ASM** does not need to be included with your application. If used, be sure to link in this module immediately after the startup code.

*This module uses the macro COMPRESSED to determine how to initialize class **FAR_DATA**.*

When far data is used, **FARDATA.ASM** inserts an initializer into the initializer table to either copy the class **FAR_DATA** from its position in EPROM, or use the Paradigm LOCATE decompression module to decompress the class. In either case, the initialization of the **FAR_DATA** class is automatic and requires no assistance from the user.

BCPP50.INC

This include file contains Borland C++-specific definitions, common to all assembly language files included with the compiler support package.

STARTUP.INC

The **STARTUP.INC** include file contains general purpose macros and definitions that can be used to streamline the assembly language startup code and helper files.

The most important of these macros is the **DefSeg** macro. This macro is used to define a segment and assign it to a class, and optionally assign it to a group. This macro doesn't put anything in the segment but it does set the order and alignment of the segment.

FARDATA.CFG

This is a Paradigm LOCATE configuration include file used to help with the handling of the **FAR_DATA** class.

Figure 9.4
FARDATA.CFG configuration file contents

```
//
// This Paradigm LOCATE configuration file is used to handle
// Borland C++ applications that use class FAR_DATA, and
// optionally compress the data.
//
#if  !defined(HASFARDATA)  // Check if application has FAR_DATA

#define _FARDATACLASSES    // Empty definitions if not used
#define _ROMFARDATACLASSES

#else

#define _FARDATACLASSES      FAR_DATA ENDFAR_DATA
#define _ROMFARDATACLASSES   ROMFARDATA ENDROMFARDATA

dup     FAR_DATA ROMFARDATA  // Copy of initialized far data

#if  defined(COMPRESSED)     // Is the data also compressed?
compress   ROMFARDATA        // Compress the ROMFARDATA class
display   compression        // Display the compression results
#endif  // defined(COMPRESSED)

#endif  // !defined(HASFARDATA)
```

If the macro **HASFARDATA** is defined, the macros **_FAR_DATA_CLASSES** and **_ROMFAR_DATA_CLASSES** are defined to be the default Borland C++ far data classes, otherwise they are defined as empty strings. If the optional macro **COMPRESSED** is defined, the class ROMFARDATA is also compressed by Paradigm LOCATE.

Run-time library helpers

This section covers the files in the HELPERS subdirectory.

Helpers are the modules required to go beyond the use of the basic compiler. With the possible exception of stack overflow checking, any C or C++ source modules you create, and don't reference the C/C++ run-time libraries, will require no additional support. While these applications are interesting, it's nice to know that the floating point, dynamic memory management, and stream I/O run-time libraries are there waiting to be put to use in your application.

Int 21h is more appropriate since we rarely have DOS in the target system.

These functions are not only ROMable, but you can't beat the price/performance ratio for these hand-optimized Borland libraries. They do need some external support since most make Int 21h calls which must be supported by the embedded system. Here we will review each major group of run-time library helpers so you can get the most out of your Borland C++ compiler.

TYPEDEFS.H

TYPEDEFS.H defines the common types used by the Borland C++ compiler support package and is used by each of the C helper modules.

DOSEMU.C DOSEMU.H

DOSEMU.C, together with **DOSEMU.H**, provides the basic Int 21h support for the Borland C++ user.

You should plan on including **DOSEMU.C** in any application using the Borland C++ run-time libraries. This Int 21h handler is always installed first and services the most simple functions; reading and writing interrupt vectors and catching unsupported operations. Any subsequent Int 21h handlers installed will use the function **_chain_intr()** to check if the service request belongs to it, otherwise passing the call to the next interrupt handler in the chain.

BCPPRTL.ASM

*Plan on including **BCPPRTL.ASM** if you use any run-time libraries.*

These are the low-level functions and data structures that are common to all Borland C++ run-time libraries. Functions for stack overflow checking, **exit()**, and **abort()** are all defined here, as are the variables **_errno** and **_doserrno**.

BCPPDMM.C
BCPPHEAP.ASM
BCPPHEAP.INC

These files are the helpers for the Borland C++ dynamic memory management functions. These functions include support for the near and far versions of **malloc()**, **free()**, **_heapcheck()**, and for those C++ users, **new** and **delete**.

Borland C++ uses a single *far heap* in the compact and large memory models, and a pair of heaps (*near heap* and *far heap*) in the small or medium memory models. The near heap is allocated in the stack segment so dynamic memory management users with a near heap should set the size of the stack to be the sum of the maximum-sized near heap and the maximum-sized stack.

The size of the heap is set in **BCPPHEAP.INC** if it is not predefined. The makefiles have a macro **HEAPSIZE** that is set to the desired heap size in KB.

BCPPSIO.C
CONSOLE.C

BCPPSIO.C is the device-independent stream I/O support module used when functions from the **printf()** and **scanf()** families are included in your application. These functions include **printf()**, **sprintf()**, **scanf()**, **sscanf()**, **puts()**, and for you C++ users, cin, and cout. You can also use **fprintf()** with this module but the **DOS_READ** or **DOS_WRITE** service code must be customized to support multiple output devices.

*Paradigm DEBUG/RT has a replacement for **CONSOLE.C** that allows the debugger to serve as the target system console.*

The module **CONSOLE.C** implements a physical device interface for the **DOS_READ** and **DOS_WRITE** functions in **BCPPSIO.C**. The functions in this module should be customized to support the serial, keyboard, or display device used in your target system.

BCPPSIO.C inserts an entry in the initializer table to have the startup code call its entry point. The initialization code hooks the Int 21h interrupt and calls the physical console initialization function in **CONSOLE.C**.

BCPPFLT.ASM
BCPPFLT.INC

These are the floating point installation helper modules. **BCPPFLT.ASM** handles the details of installing the math coprocessor emulator or support for an 80C187 (or other math hardware). The only option for user customization is the macro **FP_INT**, which selects which interrupt vector is used to handle floating point exceptions. The

default is set to 10h, which is correct for the emulator and the 80C187 chip.

While these files are required if floating point arithmetic is used, it is unlikely that they will require customization. **BCPPFLT.ASM** inserts an entry in the initializer table to have the startup code call its entry point.

FPERR.C
MATHERR.C

These are the floating point helpers built into the ROMable run-time libraries that handle exceptions. They are provided for use should you need to create custom exception handling code for your floating point application.

The function **_fperror()** in **FPERR.C** handles the intrinsic floating errors such as divide by zero and overflow. The function **__matherr()** in **_MATHERR.C** handles the exceptions from the floating point math libraries.

Configuration files

The sample applications included with Paradigm LOCATE use two different configuration files, depending on the target system. Configuration files with the .RM extension are used when developing applications to execute from EPROM or for debugging with Paradigm DEBUG and an in-circuit emulator. The configuration files with the .RT extension assume the user will be using Paradigm DEBUG/RT to debug the application from RAM.

There are only two subtle differences between the configuration files. Paradigm DEBUG/RT users don't need to use the INITCODE RESET directive since the target system is already up and running and it doesn't make sense to start at the reset vector. Chip select initialization is also omitted since the PDREMOTE/ROM target system debugging kernel usually performs this function.

The second difference deals with the changes in the memory address space. The Paradigm DEBUG/RT user must load into the RAM address space as there is no overlay memory. The MAP directives are adjusted to cover the RAM address space, mapping the remainder as

reserved. The CLASS directives are also changed to reflect addresses within the RAM address space and outside the area used by the debugging kernel.

Integrated Development Environment

If you plan to do your application development under Windows with the Borland Integrated Development Environment (IDE), you will want to check out the following development guidelines. The integrated development environment is of course designed for DOS and Windows application development, but this does not prevent one from putting it into service for the design of embedded applications.

Paradigm LOCATE comes with an IDE support package - Paradigm Addon for Borland C++ IDE. Paradigm Addon streamlines your embedded system development cycle and makes you more productive. It enables you to develop embedded applications from within the IDE. You can now create, build, and debug your applications without ever leaving the environment.

Each of the Borland C++ examples includes a pre-configured project file (having the .IDE file extension) for working under the Windows-hosted IDE, making it easy to get started. Because of the many options set by these files, we strongly recommend that new users consider taking one of these example project files as the starting point for their own application.

Installing Paradigm Addon

Paradigm Addon allows you to develop embedded applications in the IDE. However, there are a few minor details that must be considered. Be sure to go through the *Getting Started* chapter of the Paradigm Addon help topic in the IDE Online Help. It shows you how things are set up and how they work. It also has information on upgrading projects created by previous versions of Borland C++ IDE.

The Paradigm Addon is installed when you install Paradigm LOCATE for the Borland C++ 5.02 compiler. Paradigm Addon (**PARADIGM.DLL**) is enabled by the Setup program. However, it can be disabled using the Borland **ADDONREG.EXE** utility.

Compared with the setup for IDE users, the makefile approach is downright simple. Each example supplied as part of the compiler support package includes a makefile for guaranteeing the correctness of the build process, from compiling through the running of Paradigm LOCATE.

The remainder of this section documents the files, and their contents, used in the makefile approach. We welcome any new Paradigm LOCATE users to check out the contents of each makefile, and to incorporate those features which best suit the requirements of the application. It is a simple matter to add new modules to any of the makefiles, while maintaining complete control over the memory model, floating point option, and other application-dependent build options.

PARADIGM.MKF

This MAKE include file processes the makefile macro definitions to produce the list of file dependencies and compiler, assembler, linker, and Paradigm LOCATE options. It is used primarily to hide the routine complexity of application build options, letting the makefile remain focused on the application dependent issues.

Common makefile macros

Because the makefiles supplied with the Borland C++ examples are complete, we welcome you to take them and use them as the starting point for your own applications. To help you get started, here is a peek at the makefile for the example in EXAMPLES\FPDEMO. Each macro includes a short description of the build options so you can customize it to match your own target system and debugging tools.

COMPDIR

Here is how to make quick modifications and get this application built with a minimum of fussing. If you are not using the default Borland C++ directory, or it is installed on a different drive, change the macro **COMPDIR** to match your system. This will allow the makefile to create a custom **TURBOC.CFG** file with the compiler options and path for include files.

COMPCFG This is the name of Borland C++ compiler configuration file. The makefile will create this file any time a compiler option changes. This file is used to make sure that another compiler configuration file containing undesired compiler options is not used.

MKF This macro defines the filename of the makefile and is used to guarantee a complete rebuild of the application should the makefile be changed.

MODEL This macro is set to select the desired memory model. You can usually use any memory model, but there may be restrictions on the use of huge model with some run-time library functions.

CPU The **CPU** macro is used to select a code generator for the target system. Set **CPU** to zero if using an 8086/88 microprocessor, or one if using a NEC V-Series, 80186-family or higher microprocessor.

DEBUG This option allows you to create three different output file types. Setting **DEBUG** to 0 will select the *.RM configuration file and disables all debug information, making for the fast builds (and the slowest debugging sessions). Setting **DEBUG** to 1, will also select the ***.RM** configuration file but will enable compiling and assembling with full debug information. Setting **DEBUG** to 2 also enables debug information but the ***.RT** configuration file will be used.

OPTIMIZE This macro can be used to select the desired level of optimizations. Set to zero for minimal optimizations during development and to one if you want to optimize for code size. Set **OPTIMIZE** to two if you want to optimize for your application for speed. The **PARADIGM.MKF** file can be edited to support other, more specific, optimizations supported by Borland C++.

WARNINGS Use this macro definition to enable or disable compiler warnings. Specific warning control can be specified by adding **-wxxx** command line options to the **TURBOC.CFG** file definition found at the end of the makefile.

CODESTRING Borland C++ supports placing string literals in the code segment with the **-dc** command line option. You can enable this feature by setting

CODESTRING to 1. Remember to use far pointers when string literals are in the code segment.

DUPSTRING Setting **DUPSTRING** to one is used to force the compiler to merge duplicate strings, an excellent idea if you use lots of string literals and you want only a single instance.

CHECKSTACK This option is used to enable or disable the compiler stack checking logic. During development it is not a bad idea to turn this option on to catch errors that might otherwise crash a target system.

FLOAT Set the **FLOAT** macro to zero to select the option for no floating point support. If you want to use the math coprocessor emulation included in Borland C++, set **FLOAT** to 2. If you have a real math coprocessor in your target system, set **FLOAT** to 3.

FARDATA

If FARDATA is 1, the makefile defines the macro HASFARDATA. If FARDATA is 2, the makefile also defines the macro COMPRESSED.

The **FARDATA** macro selects between the various levels of Borland C++ far data support. When **FARDATA** is zero, the far data support is disabled and not used. When **FARDATA** is set to 1, the application can use far data and the class **FAR_DATA** will be duplicated and placed in EPROM for copying to RAM by the initializer in the file **FARDATA.ASM**. Setting **FARDATA** to 2 will enable the use of far data, but it will be duplicated and compressed to minimize the EPROM footprint of the class **FAR_DATA**.

IOSTREAMS Set the **IOSTREAMS** macro to zero to disable use of C/C++ stream I/O. When set to 1, use of memory formatting run-time libraries are supported (sprintf, etc). Set this to 2 to select full C/C++ I/O stream support.

EXCEPTIONS The **EXCEPTIONS** macro is used to enable or disable the use of exception handling. When disabled, the appropriate Borland **NOEH?.LIB** library will be linked in, minimizing the size of the application.

STACK This option lets you set the size of the default stack in the startup code. The stack will be in DGROUP for the small and compact memory models, and in its own class in the compact, large, and huge memory models.

The stack size can be set to any value but the minimum size is likely to be 256 bytes. If you have many levels of function nesting, with lots of automatics, or use the floating point emulation, the stack will need to be larger. If the stack and heap are sharing the stack segment, the stack must be set to the maximums of the desired stack and heap.

HEAPSIZE

The **HEAPSIZE** macro sets the size of the heap, or it disables the run-time library heap functions.

The Borland C++ far heap is used as the default heap in compact, large, and huge memory models. In the small or medium memory models, the default heap is part of the stack (set the stack size to include your desired heap size) and **HEAPSIZE** declares the size of the far heap.

Sample applications

This section covers the files in the EXAMPLES subdirectory.

This section covers the sample applications included with the Borland C++ support package. Except for the simple application found in DEMO, all examples have complete makefiles that permit easy setting of the memory model, floating point option, **FAR_DATA** class handling, and output files for burning EPROMs or debugging with Paradigm DEBUG.

Each example is supplied to work with the Windows-hosted IDE or the command line tools, allowing whichever development environment you are most comfortable with to be used.

DEMO

This is a simple demo, very similar to the example presented in chapter 2. The main difference is that this example includes support for building the C source file and startup code.

This is an excellent place to begin if you are just getting started with Borland C++ and embedded systems.

DMMDEMO

DMMDEMO is a simple example demonstrating the use of the Borland C++ dynamic memory management routines. In this

example, an array of pointers to random-sized character arrays is maintained. If a location has already been allocated, it is released, otherwise it is filled with data. After each allocation request, the function **heapcheck()** is used to verify the integrity of the heap.

The method used to implement the far heap optimizes the available RAM but has the disadvantage that LOCATE can't perform error checking on the class.

The default memory model for this application is small so a near heap is used. If you change to compact or large memory model, the macro **HEAPEND** in the file **BCPPDMM.C** must be changed to reflect the paragraph address of the end of the far heap in your target system.

Also note that no other classes may follow class FARHEAP in the ORDER directive. This is required as Paradigm LOCATE does not know the true size of the far heap and will not catch the overlap between the segments.

FPDEMO

This is an example designed for users of the Borland C++ floating point run-time libraries. It endlessly performs floating point calculations, checking the results for accuracy. The sample code also forces errors, divide by zero, square root of a negative number, to verify the exception handling logic works as expected.

STDIO

STDIO.C exercises the input and output streams of Borland C++. Running the ubiquitous Sieve demo, it calculates the number primes and displays the result.

Paradigm DEBUG users can inspect 'outbuf' to see the output.

If you have a UART or an LCD display, modify the file **CONSOLE.C** to work your physical device. Since we assume nothing about the target system, all input and output are with memory buffers in the default version.

CPPDEMO

Besides being a C++ application, this example does everything: floating point, C++ streams, and dynamic memory management with **new**. Check out this example if you plan to use C++ or need access to a makefile with support for all run-time library options.

COMPRESS

This example demonstrates the use of data compression with Paradigm LOCATE. The file **COMPRESS.C** allocates a number of

initialized and uninitialized data structures in the class FAR_DATA and then checks that they have been correctly initialized by the startup code.

You can build the example in any memory model and can switch between copied far data (**FARDATA** = 1) and decompressed far data (**FARDATA** = 2). Note that the decompression code requires at least 5KB of stack space to work correctly.

CRCDEMO

This example demonstrates the use of the checksum capabilities supplied with Paradigm LOCATE. Paradigm LOCATE includes complete support for IBM PC ROM BIOS extensions, and for more critical applications, the CRC16 and CRC32 algorithms. This example is an ideal starting point for those applications that need some form of redundancy checking.

In the CRCDEMO example, three separate regions of the address space are defined and the ROMBIOS, CRC16, and CRC32 checksums are computed by Paradigm LOCATE and stored immediately following each region. As the sample code executes, it recomputes each checksum or CRC and verifies that no errors have been detected.

EHDEMO

This example demonstrates the use of the C++ exception handling capabilities of Borland C++. Paradigm LOCATE fully supports the use of exception handling so your application can get maximum benefit from this technology.

For those applications that need to manipulate strings, this example also uses the C++ string class.

Borland C++ requires a far heap to use exception handling. You will get a linker error the first time you try to build this application. Just set the **HEAPEND** macro in **BCPPDMM.C** and you will be on your way.

NURAM

This example demonstrates the handling of non-volatile data in an embedded application. Data placed in a non-volatile segment is untouched by the startup code and is only modified by the application.

CONST

This example demonstrates the use of constant data in a read-only address space. By default, the compiler places all initialized data in the default segment. By declaring the data as 'const' and '__far', it can be moved out and stored in flash or EPROM. This technique reduces the RAM footprint of an application.

C H A P T E R

10

Microsoft C/C++ guide

This chapter is a comprehensive overview to using Microsoft C/C++ with embedded system applications. Covered in detail are the individual components of the Microsoft C/C++ compiler support package, including the startup code, run-time library helper functions, and example applications.

An understanding of the Microsoft C/C++ memory models is also very useful.

If you are just getting started with Microsoft C/C++, you will need to be familiar with the Microsoft compiler, macro assembler, LINK, and NMAKE utilities in order to use this chapter. For those that would rather use the Visual C++ Workbench as a development environment, complete instructions for using this powerful, Windows-hosted software development environment are available. Also covered is a complete description of the standard makefiles shipped with each example in the Microsoft C/C++ compiler support package.

Those readers who have worked with Microsoft C/C++ in past, should have little trouble migrating to using it as the centerpiece of an embedded system application. The Microsoft C/C++ compiler brings the best of PC application development tools to the embedded system user, at a price/performance ratio not matched by traditional embedded system development tools.

Startup code

This section covers the files in the MSC80 root directory.

NOTE: Because the startup module defines the order and alignment of segments, it MUST be linked in first.

The startup code files take an application from the reset vector to the start of your application. Before **main()** can be handed control of the processor, the application stack and segment registers must be initialized, data copied from EPROM to RAM, and the run-time libraries installed and initialized. Even though a lot of work must be accomplished to prepare an application for execution, you rarely will need to modify the startup code. Using the Paradigm LOCATE INITCODE directive and the startup code initializers, applications can add extensions without modifying any of the Paradigm-supplied startup code.

MSC80.ASM

If you stop and take a peek inside the startup code you will find the following organization. Following the opening comments is a series of **DefSeg** macros that define the default set of segments/classes used by a Microsoft C/C++ embedded application.

*Figure 10.1 Segment ordering and alignment with **DefSeg**.*

```
;
; Segment and group declarations.  The order of these
; declarations is used to control the order of segments in the
; .ROM file since the Microsoft linker copies segments in the
; same order they are encountered in the object files.
;

DefSeg _TEXT, para, public, CODE,  <>     ; Default code
DefSeg _DATA, para, public, DATA,  DGROUP ; Initialized data
DefSeg xiheap,word, common, DATA,  DGROUP ; Heap initialization
DefSeg CONST, word, public, CONST, DGROUP ; Constant data
DefSeg HDR,   word, public, MSG,   DGROUP ; Misc. data
DefSeg _BSS,  word, public, BSS,   DGROUP ; Uninitialized data
DefSeg _STACK,para, stack,  STACK, DGROUP ; Program stack
```

If you were to take a closer look and examine the definition of **DefSeg** (in the file **STARTUP.INC**), you would discover that these macros don't define any code or data, they simply open and immediately close the segment. Instead of defining code or data, the position of **DefSeg** macros is used by the startup code to set the order and alignment of the default segments and classes required by a Microsoft C/C++ application.

Following the segment ordering code is the startup code entry point, which also happens to be the start of the application code. At reset, the startup code must take care of the following chores in order to jump start the application:

1. stack initialization
2. copy the initialized data from EPROM to RAM
3. clear the uninitialized data
4. execute any initializers/constructors present
5. call **main()**

Steps 1 and 5 should be obvious and need no further comment. Steps 2 and 3 handle the initialization of the classes DATA and BSS which are defined by Microsoft C/C++ to contain the initialized and uninitialized data for the application. Since initialized data can only exist in EPROM when power is first applied to the system, step 2 copies any initialized data from its position in EPROM to its position in RAM. The size of the memory block to be copied is determined by measuring the distance from the start of class DATA to the start of class BSS.

Uninitialized data refers to the C language declaration. The data is actually initialized to zero.

Step 3 fills the class BSS, which contains all the uninitialized static data in your application, with zero bytes. The size of the BSS class is determined by the start of the class STACK.

The work done by step 4 can seem rather complicated but it is really quite simple. Our goal is to automatically initialize a run-time library package without requiring source code modification. Every run-time library package or C++ object requiring initialization places a pointer into the segment containing an array of these initializers. The startup code then executes each initializer in order of priority before calling **main()**. If the module is linked in, its initializer will be called and we never have to modify any source code.

Finally, you may be wondering where the peripheral register initializations for chip selects, wait states, and DRAM refresh are located in the startup code. This is handled by the Paradigm LOCATE INITCODE directive so there is no need to modify the startup code to adjust for different target systems. Here is a Paradigm LOCATE configuration file fragment showing the Intel 80C186EA/XL chip select initialization code.

Figure 10.2 INITCODE reset and chip select register initialization

```
initcode  reset           \     // Reset vector
          umcs = 0xf838   \     // 32KB EPROM
          lmcs = 0x0ff8         // 64KB RAM
```

The startup code not only initializes the stack, but also defines the size of the stack using the macro STKSIZE.

CINIT.ASM

This module handles the classes **FAR_DATA** and **FAR_BSS**, if they are present in your application, plus the startup code initializers.

*This module uses the macro **COMPRESSED** to determine how to initialize class FAR_DATA.*

When far data is used, **CINIT.ASM** will copy the class **FAR_DATA** from its position in EPROM, or use the Paradigm LOCATE decompression module to decompress the class. The memory used in the class **FAR_BSS** is also set to zero, as required by the compiler. In either case, the initialization of the **FAR_DATA** class is automatic and requires no assistance from the user.

Following the initialization of memory, the module will call any linked-in initializers. Initializers are broken down into four groups. The Paradigm initializers are used to install support for run-time library helpers. These are followed by the Microsoft near/far initializers. The last group of initializers are used by C++ applications to call the constructors for any global objects.

MSC80.INC

This include file contains Microsoft C/C++-specific definitions, common to all assembly language files included with the compiler support package.

STARTUP.INC

The **STARTUP.INC** include file contains general purpose macros and definitions that can be used to streamline the assembly language startup code and helper files.

The most important of these is the **DefSeg** macro. This macro is used to define a segment and assign it to a class, and optionally assign it to a group. This macro doesn't put anything in the segment but it does set the order and alignment of the segment.

FARDATA.CFG

This is a Paradigm LOCATE configuration include file used to help with the handling of the **FAR_DATA** and **FAR_BSS** classes.

Figure 10.3
FARDATA.CFG
configuration file contents

```
//
//  This Paradigm LOCATE configuration file is used to handle
//  Microsoft C/C++ applications that use class FAR_DATA, and
//  optionally compress the data.
//
//  This file has no effect if class FAR_DATA is not used.
//
#if  !defined(HASFARDATA)           // Check if FAR_DATA exists

#define  _START_RAM_DATA  DATA      // Class DATA is first

#define  _FAR_DATA_CLASSES          // Empty if not used
#define  _ROMFAR_DATA_CLASSES

#else

#define  _START_RAM_DATA   FAR_DATA // Class FAR_DATA is first

#define  _FAR_DATA_CLASSES FAR_DATA ENDFAR_DATA FAR_BSS
ENDFAR_BSS
#define  _ROMFAR_DATA_CLASSES   ROMFARDATA ENDROMFARDATA

dup      FAR_DATA ROMFARDATA       // Copy initialized far data

#if  defined(COMPRESSED)           // Is the data compressed?
compress  ROMFARDATA               // Compress ROMFARDATA class
display   compression              // Display compression
results
#endif   // defined(COMPRESSED)
#endif   // !defined(HASFARDATA)
```

If the macro **HASFARDATA** is defined, the macros **_FAR_DATA_CLASSES** and **_ROMFAR_DATA_CLASSES** are defined to be the default Microsoft C/C++ far data classes, otherwise they are defined as empty strings. If the optional definition **COMPRESSED** is defined, the class ROMFARDATA is compressed by Paradigm LOCATE.

Run-time library helpers

This section covers the files in the HELPERS subdirectory.

Helpers are the modules required to go beyond the use of the basic compiler. With the possible exception of stack overflow checking, any

C or C++ source modules you create and don't reference the C/C++ run-time libraries, will require no additional support. While these applications are interesting, it's nice to know that the floating point, dynamic memory management, and stream I/O run-time libraries are there waiting to be put to use in your application.

Int 21h is more appropriate since we rarely have DOS in the target system.

These functions are not only ROMable, but you can't beat the price/performance ratio for these hand-optimized Microsoft libraries. They do need some external support since most make Int 21h calls which must be supported by the embedded system. Here we will review each major group of run-time library helpers so you can get the most out of your Microsoft C/C++ compiler.

TYPEDEFS.H

TYPEDEFS.H defines the common types used by the Microsoft C/C++ compiler support package and is used by each of the C helper modules.

DOSEMU.C
DOSEMU.H

DOSEMU.C, together with **DOSEMU.H**, provides the basic Int 21h support for the Microsoft C/C++ user.

You should plan on including **DOSEMU.C** in any application using the Microsoft C/C++ run-time libraries. This Int 21h handler is always installed first and services the most simple functions; reading and writing interrupt vectors and catching unsupported operations. Any subsequent Int 21h handlers installed will use the function **_chain_intr()** to check if the service request belongs it, otherwise passing the call to the next interrupt handler in the chain.

MSCRTL.ASM

Plan on including MSCRTL.ASM if you use any run-time libraries.

These are the low-level functions and data structures that are common to all Microsoft C/C++ run-time libraries. Functions for stack overflow checking, **exit()**, and **abort()** are all defined here, as are the variables **_errno** and **_doserrno**.

MSCRTL.ASM also provides *near heap* support. The macro **NHEAPEND** determines the size of the *near heap*.

MSCDMM.C
MSCHEAP.ASM
MSCHEAP.INC

These files are the helpers for the Microsoft C/C++ dynamic memory management functions. These functions include support for the near and far versions of **malloc()**, **free()**, **_heapchk()**, and for those C++ users, **new** and **delete**.

Microsoft C/C++ supports both a *far heap* and a *near heap*. The near heap is allocated immediately following the stack. You can set the size of the near heap, which is part of the group DGROUP, by changing the definition of **NHEAPSIZE** in the file **MSCRTL.ASM**.

The size of the heap is set in **MSCHEAP.INC** if it is not predefined. The makefiles have a macro **HEAPSIZE** that is set to the desired heap size in KB.

MSCSIO.C
CONSOLE.C

MSCSIO.C is the device-independent stream I/O support module used should functions from the **printf()** and **scanf()** families be included in your application. These functions include **printf()**, **sprintf()**, **scanf()**, **sscanf()**, **puts()**, and for you C++ users, cin, and cout. You can also use **fprintf()** with this module but the **DOS_READ** or **DOS_WRITE** service code must be customized to support multiple output devices.

*Paradigm DEBUG/RT has a replacement for **CONSOLE.C** that allows the debugger to serve as the target system console.*

The module **CONSOLE.C** implements a physical device interface for the **DOS_READ** and **DOS_WRITE** functions in **MSCSIO.C**. The functions in this module should be customized to support the serial, keyboard, or display device used in your target system.

MSCSIO.C inserts an entry in the initializer table to have the startup code call its entry point. The initialization code hooks the Int 21h interrupt and calls the console initialization function in **CONSOLE.C**.

MSCFLT.ASM

This is the floating point installation helper module. **MSCFLT.ASM** handles the details of installing the math coprocessor emulator or support for an 80C187 (or other math coprocessor hardware). The only option for user customization is the macro **FP_INT**, which selects which interrupt vector is used to handle floating point exceptions. The default is set to 10h, which is correct for the emulator and the 80C187 chip.

For Microsoft floating point emulation, it is required to set the target processor register bit that causes ESC opcode to be trapped. On 186 processors, set the ET bit in RELREG. On 386 processors, set the EM bit in CR0. On some processors including 188 and V25, this is automatic. Sample code to enable ESC opcode trap is included in **MSCFLT.ASM**. See documentation inside the file for details.

While this file is required if floating point arithmetic is used, it is unlikely that they will require customization. **MSCFLT.ASM** inserts an entry in the initializer table to have the startup code call its entry point.

Note! If you are using the Microsoft emulated floating point library, you must set the processor control register bit that causes ESC opcodes to be trapped. On 186 processors, set the ET bit in RELREG. On 386 processors, set the EM bit in CR0. **DOSEMU.C** must be linked to handle the Int 7 exception. On some processors, this is automatic (188, V25 etc).

Configuration files

The sample application included with Paradigm LOCATE uses two different configuration files, depending on the target system. Configuration files with the .RM extension are used when developing applications to execute from EPROM or for debugging with Paradigm DEBUG and an in-circuit emulator. The configuration files with the .RT extension assume the user will be using Paradigm DEBUG/RT to debug the application from RAM.

There are only two subtle differences between the configuration files. Paradigm DEBUG/RT users don't need to use the INITCODE RESET directive since the target system is already up and running and it doesn't make sense to start at the reset vector. Chip select initialization is also omitted since the PDREMOTE/ROM target system debugging kernel usually performs this function.

The second difference deals with the changes in the memory address space. The Paradigm DEBUG/RT user must load into the RAM address space as there is no overlay memory. The MAP directives are adjusted to cover the RAM address space, mapping the remainder as reserved. The CLASS directives are also changed to reflect addresses

within the RAM address space and outside the area used by the debugging kernel.

Visual Workbench

The major problem of using the Microsoft Visual Workbench (VWB) to develop embedded applications is the assumption that users want to create a PC-style application. That is, the Visual Workbench always tries to bring in the default startup code and default libraries. As many of the default compiler/linker options are not appropriate for embedded applications, steps must be taken to ensure the ROMable startup code and run-time libraries are used in building our application.

Besides not supporting MASM directly, the Visual Workbench does not provide users with full control over the compiler, linker, and NMAKE, making the Visual Workbench project files incompatible with embedded applications. Fortunately, the Visual Workbench does accept external makefiles, a capability we can exploit to use the Visual Workbench as the basis for embedded system application development.

Setting up the Visual Workbench

Generally speaking, no special settings are needed because we are not using the internal project files created by the VMB. Instead, the makefile associated with each of the Paradigm examples will serve as the external project file. Let's take the CPPDEMO application as an example and see how easy it is to setup the Visual Workbench to build this application using the supplied makefile.

Creating a project

Select **P**roject|**O**pen to load the application makefile into the Visual Workbench as an external project.

```
File Name: makefile
[x]   Use as an External Makefile
```

If the MAKEFILE is not in the current directory, use the directory tree to go to where it is located.

External project options

The External Project Option dialog window will pop up as soon as Open Project dialog window is closed. In this window, you can enter the target filename and select target file type.

```
Debug Target Name:   CPPDEMO.AXE
Project Type:        Other
Debug Build:         NMAKER /f MAKEFILE
Release Build:       NMAKER /f MAKEFILE
```

This tells the project manager that **CPPDEMO.AXE** is the final target to be built by the Visual Workbench and that it is not a DOS or Windows application.

Adding tools

Paradigm DEBUG can also be integrated into the Visual Workbench tool set so it can be started from the Tools pull-down menu. Select the Options|Tools menu item and fill the fields to install Paradigm DEBUG, as shown below.

```
[Add]    (Use the directory tree to select Paradigm DEBUG)

Command Line:          C:\PD\PDRT186.PIF
Menu Text:             Paradigm &DEBUG
Arguments:             CPPDEMO.AXE
Initial Directory:     (your current directory)
```

If you do not want to pass the .AXE file name in command line, you can leave *Arguments* field blank. You can also control if you want to be prompted for arguments by enabling or disabling the *Ask for arguments* field.

Setting up the environment

These final options are set to match your system and should be adjusted accordingly.

```
Options:Directories:   "Directories" dialog
Executable Files Path  c:\msvc\bin;c:\locate
```

Starting your own project

It is strongly recommended to use one of our example makefiles as a template for your own project. This ensures that correct settings for both the Paradigm and Microsoft tools and embedded system targets. To customize the makefile for use with your own application, all that is required is to add or delete modules to the source and object file dependencies.

Makefiles

Each example supplied as part of the compiler support package includes a makefile for guaranteeing the correctness of the build process. We welcome any new Paradigm LOCATE users to check out the contents of each makefile, and to incorporate those features which best suit the requirements of the application. It is a simple matter to add new modules to any of the makefiles, while maintaining complete control over the memory model, floating point option, and other application-dependent build options.

PARADIGM.MKF

This **MAKE** include file processes the makefile macro definitions to produce the list of file dependencies and compiler, assembler, linker, and Paradigm LOCATE options. It is used primarily to hide the routine complexity of application build options, letting the makefile remain focused on the application dependent issues.

Common makefile macros

Because the makefiles supplied with the Microsoft C/C++ examples are complete, we welcome you to take them and use them as the starting point for your own applications. To help you get started, here is a peek at the makefile for the example in EXAMPLES\FPDEMO. Each macro describes the build options you can customize to meet the requirements of your target system and debugging tools.

COMPDIR

Here is now to make quick modifications and get this application built with a minimum of fussing. If you are not using the default Microsoft C/C++ directory, or it is installed on a different drive, change the macro **COMPDIR** to match your system.

MKF

This macro defines the filename of the makefile and is used to guarantee a complete rebuild of the application should the makefile be changed.

MODEL

This macro is set to select the desired memory model. You can usually use any memory model, but there may be restrictions on the use of huge model with some run-time library functions.

CPU The **CPU** macro is used to select a code generator for the target system. Set **CPU** to zero if using a 8086/88 microprocessor, one if using an 80186-family or NEC V-Series microprocessor, or two for a 80286 or higher.

DEBUG This option allows you to create three different output file types. Setting DEBUG to 0 will select the *.RM configuration file and disable all debug information, making for the fast builds (and the slowest debugging sessions). Setting **DEBUG** to 1, will also select the ***.RM** configuration file but will enable compiling and assembling with full debug information. Setting **DEBUG** to 2 also enables debug information but the ***.RT** configuration file will be used.

WARNINGS This option is used to select the desired level of compiler syntax checking. Set this macro to the level which matches your coding style and development requirements.

OPTIMIZE This macro can be used to select the desired level of optimizations. Set to zero for minimal optimizations during development and to one if you want to optimize for size, or two if you want to optimize for speed. The **PARADIGM.MKF** file can be edited to support other, more specific, optimizations supported by Microsoft C/C++.

CHECKSTACK This option is used to enable or disable the compiler stack checking logic. During development it is not a bad idea to turn this option on to catch errors that might otherwise crash a target system.

FLOAT Set the **FLOAT** macro to zero to select the option for no floating point support. If you want to use the alternate math library, set **FLOAT** to 1. If you want to use the math coprocessor emulation included in Microsoft C/C++, set **FLOAT** to 2. If you have a real math coprocessor in your target system, set **FLOAT** to 3.

If you are using FLOAT = 2, see the note regarding ESC opcode trapping on page 130

FARDATA The **FARDATA** macro selects between the various levels of Microsoft C/C++ far data support. When **FARDATA** is zero, the far data support is disabled and not used. When **FARDATA** is set to 1, the application can use far data and the class **FAR_DATA** will be duplicated and placed in EPROM for copying to RAM by the initialization code in the file **CINIT.ASM**. Class **FAR_BSS** will also

*If **FARDATA** is 1, the makefile defines the macro **HASFARDATA**. If **FARDATA** is 2, the makefile also defines the macro **COMPRESSED**.*

be defined and the startup code will fill this class with zero. Setting **FARDATA** to 2 will enable the use of far data, but it will be duplicated and compressed to minimize the EPROM footprint of the class **FAR_DATA**.

IOSTREAMS Set the **IOSTREAMS** macro to zero to disable use of C/C++ stream I/O. When set to 1, use of memory formatting run-time libraries are supported (sprintf, etc). Set this to 2 to select full C/C++ I/O stream support.

STACK This option lets you set the size of the default stack in the startup code.

The stack size can be set to any value but the minimum size is likely to be 256 bytes. If you have many levels of function nesting, with lots of automatics, or use the floating point emulation, the stack will need to be larger.

HEAPSIZE The **HEAPSIZE** macro sets the size of the heap, or it disables the run-time library heap functions.

The Microsoft C/C++ far heap is used as the default heap in compact, large, and huge memory models. In the small or medium memory models, the default heap is part of the stack (set the stack size to include your desired heap size) and **HEAPSIZE** declares the size of the far heap.

Sample applications

This section covers the files in the EXAMPLES subdirectory.

This section covers the sample applications included with the Microsoft C/C++ support package. Except for the simple application found in DEMO, all examples have complete makefiles that permit easy setting of the memory model, floating point option, **FAR_DATA** and **FAR_BSS** class handling, and output files for burning EPROMs or debugging with Paradigm DEBUG.

Each example is supplied to work with the Windows-hosted Visual Workbench or the command line tools, allowing whichever development environment you are most comfortable with to be used.

DEMO

This is a simple demo, very similar to the example presented in chapter 2. The main difference is that this example includes support for building the C source file and startup code.

This is an excellent place to begin if you are just getting started with Microsoft C/C++ and embedded systems.

DMMDEMO

DMMDEMO is a simple example demonstrating the use of the Microsoft C/C++ dynamic memory management routines with near and far heaps. In this example, an array of pointers to random-sized character arrays is maintained. If a location has already been allocated, it is released, otherwise it is filled with data. After each allocation request, the function **_heapchk()** is used verify the integrity of the heap.

The method used to implement the far heap optimizes the available RAM but has the disadvantage that LOCATE can't perform error checking on the class.

In order for the far heap to operate properly, the macro **FHEAPEND** in the file . **MSCDMM.C** must be changed to reflect the paragraph address of the end of the far heap in your target system. Also note that the class STACK, which contains the far heap, must not be followed by any other classes as the size of the far heap is not fully known by Paradigm LOCATE.

FPDEMO

If you plan to use emulated floating point, see the note regarding ESC opcode trapping on page 130.

This is an example designed for users of the Microsoft C/C++ floating point run-time libraries. It endlessly performs floating point calculations, checking the results for accuracy. The sample code also forces errors, divide by zero, square root of a negative number, to verify the exception handling logic works as expected.

STDIO

STDIO.C exercises the input and output streams of Microsoft C/C++. Running the ubiquitous Sieve demo, it calculates the number primes and displays the result.

Paradigm DEBUG users can inspect 'outbuf' to see the output.

If you have a UART or an LCD display, modify the file **CONSOLE.C** to work your physical device. Since we assume nothing about the target system, all input and output are with memory buffers in the default version.

CPPDEMO

Besides being a C++ application, this example does everything: floating point, C++ streams, and dynamic memory management with **new**. Check out this example if you plan to use C++ or need access to a makefile with support for all run-time library options.

COMPRESS

This example demonstrates the use of data compression with Paradigm LOCATE. The file **COMPRESS.C** allocates a number of initialized and uninitialized data structures in the classes FAR_DATA and FAR_BSS and then checks that they have been correctly initialized by the startup code.

You can build the example in any memory model and can switch between copied far data (**FARDATA** = 1) and decompressed far data (**FARDATA** = 2). Note that the decompression code requires at least 5KB of stack space to work correctly.

CRCDEMO

This example demonstrates the use of the checksum capabilities supplied with Paradigm LOCATE. Paradigm LOCATE includes complete support for IBM PC ROM BIOS extensions, and for more critical applications, the CRC16 and CRC32 algorithms. This example is an ideal starting point for those applications that need some form of redundancy checking.

In the CRCDEMO example, three separate regions of the address space are defined and the ROMBIOS, CRC-16, and CRC-32 checksums are computed by Paradigm LOCATE and stored immediately following each region. As the sample code executes, it recomputes each checksum or CRC and verifies that no errors have been detected.

NURAM

This example demonstrates the handling of non-volatile data in an embedded application. Data placed in a non-volatile segment is untouched by the startup code and is only modified by the application.

CONST

This example demonstrates the use of constant data in a read-only address space. By default, the compiler places all initialized data in the default segment. By declaring the data as 'const' and '__far', it can be moved out and stored in flash or EPROM. This technique reduces the RAM footprint of an application.

A P P E N D I X

A

Warning diagnostics

The warnings listed in this appendix indicate potential problems or relay diagnostic information to the user concerning the translation process. Each warning message is listed in numerical order and may be disabled by a command line option or in the configuration file, if you prefer to ignore the warning.

Paradigm LOCATE warnings

Message explanations

The following warning diagnostics are produced by Paradigm LOCATE while the processing the input files, command line arguments, or configuration file.

W1000 **No address assigned to segment *'seg/class'***

The identified segment did not appear in a CLASS, SEGMENT or ORDER directive and no physical address assignment has been made, leaving the segment to start at address 0x00000.

W1001 **Unable to translate debug info for *'module':'symbol'***

Paradigm LOCATE does not support the translation of the type information for the symbol and the type information is lost from the debug records.

W1002 Assumed absolute symbol *'name'*

Paradigm LOCATE failed to successfully translate the segment address for the specified symbol. While this can indicate a problem, it is very likely that the symbol is already an absolute address and no address translation is possible.

W1003 Segment constant is larger than 16-bits in *'file'*, line *'nnn'*

The physical address assigned to a segment or class cannot be represented as a 16-bit unsigned integer and has been truncated. Segment fixups should have values between 0x0000 and 0xFFFF.

W1004 Address *'addr'* is large in *'file'*, line *'nnn'*

The specified address in the configuration file directive is too large to be represented as a 20-bit unsigned integer and has been truncated.

W1005 Output data truncated in *'file'*, line *'nnn'*

The output data used in the INITCODE I/O port output argument is larger than 0xFFFF and has been truncated.

W1006 Linker output files have different creation times

The file dates and times for the linker output are different. This warning may indicate that the relocatable load module (.ROM) and the corresponding map file (.MAP) were not produced at the same time. This warning can also occur when a post-processing utility is used to process the relocatable load module before running Paradigm LOCATE.

W1007 Segment *'seg'* lacks a class name

The segment *seg* has been declared without a class name. This segment can only have a physical address assigned using the SEGMENT directive.

W1008 Multiple address assignments made to class *'name'*

The class *name* appears in two or more CLASS or ORDER directives. Paradigm LOCATE only recognizes the first address assignment made to a class.

W1009 **Multiple address assignments made to segment *'seg'***
The identified segment appears in two or more SEGMENT directives. Paradigm LOCATE only recognizes the first address assignment made to a segment.

W1010 ***'class'* in multiple DUP directives in *'file'*, line *'nnn'***
Paradigm LOCATE has found a class named in multiple DUP directives, perhaps indicating a configuration file problem.

W1011 ***'class'* in multiple COMPRESS directives in *'file'*, line *'nnn'***
The named class has turned up in multiple **COMPRESS** directives, where only the first directive is effective.

W1012 **Alias between segments *'seg/class'* and *'seg/class'***
In the event of an alias, Paradigm LOCATE will use the address of the first non-zero length segment.
Two or more segments in different classes share a common segment fixup and the configuration file directives have assigned unique physical addresses. This makes the segment translation process for these segments ambiguous and it is possible for a fixup to be incorrectly computed. This warning is usually the result of a zero length segment ending a class.

W1013 **Overlap between segments *'seg/class'* and *'seg/class'***
The memory address spaces for the two named segments intersect, causing one segment to overlap the other. This warning is most likely due to a segment growing into another segment or an error in the configuration file address assignments.

W1014 **Segment *'seg/class'* exceeds the 1MB address space**
The length of the segment seg in class *'class'* extends it beyond the end of the 1MB address space, preventing all or part of the segment from being addressed.

W1015 **Reserved region violation by segment *'seg/class'***
All or part of the specified segment is located in a region of the memory address space that has been marked as reserved using the MAP directive.

W1016 **Overlap between regions at *'addr'* and *'addr'***
Two regions defined in configuration file MAP directives share a common portion of the memory address space yet have different memory access attributes.

W1017 **Segment *'seg/class'* is mapped to multiple address spaces**
The segment '*seg'* in class *'class'* spans two separate regions of the memory address space having different memory access attributes.

W1018 **Intel OMF86 does not support register variables**
Many compilers can disable the use of register variables.
Intel OMF86 debug information does not support the use of register variables and the debug information was lost. If you are using a debugger or in-circuit emulator and wish to see the variables assigned to registers as part of the debug record, you must disable the use of register variables by your compiler or assembler.

W1019 **Intel OMF86 does not support object languages**
Intel OMF86 does not support languages like C++ or Object Pascal and object-related debugging information may have been lost.

W1020 **Intel OMF86 does not support register parameters**
Pass parameters on the stack when using Intel OMF86 files.
Intel OMF86 does not support parameters to functions and procedures to be passed in registers and the debug information was lost.

W1021 **Intel OMF86 does not support based pointers**
Intel OMF86 debug information does not support the use of based pointers and the debug information was lost.

W1022 **Intel OMF86 does not support inline functions**
Disable inline functions while debugging.
Intel OMF86 does not support inline functions and the debug information was lost.

W1023 **Unsigned 32-bit value truncated to 24-bits**
Intel OMF86 does not have support for 32-bit unsigned integers and the corresponding debug information was truncated to 24-bits.

W1024 **Ambiguous structure detected - type information lost**
Paradigm LOCATE is unable to determine the size of a structure and the debug information for the structure has been lost. This warning is

caused by insufficient debugging information being available, often when unnamed structure members are used.

W1025 **Ambiguous type reference in function *'name'***

Due to a lack of debug information output by the compiler, the parameter names and types for the function *name* have been lost. There isn't anything you can do but disable this warning should it occur.

W1026 **Type index too large (*'index'*) - type info lost**

A type index greater than 07FFFH has been detected in the output and has been eliminated. This warning is most likely due to an error in the debugging information or more type records than are supported by Intel OMF86.

W1027 **Illegal type index detected for *'symbol'***

The named symbol has a type index larger than the maximum defined for the module and has been eliminated from the debug information. This warning is caused by an error in the debug information.

W1028 **Too many line number records in module 'name'**

The number of line number records in the module exceed the capabilities of Paradigm LOCATE and have been lost. To correct this problem, split the offending source module into two or more parts and rebuild the application.

W1030 **No 'type' output was written to 'file'**

You probably need to add OFFSET=0x????? to your HEXFILE directive.

This warning diagnostic occurs when an EPROM output file was requested but no data was found in the region defined by the base address and size of the EPROM. This warning is most likely due to the failure to include the segments in the address space of the EPROM image in an OUTPUT directive or the failure to define a suitable offset and size for extracting the EPROM image.

W1031 **Requested *'type'* output exceeds 1MB address space**

You are creating a file that exceeds the 1MB address space boundary. Adjust the SIZE, OFFSET, and/or SPLIT parameters to stay within the 1MB address space.

W1032 Segment *'seg/class'* is output to a *memtype* region
Paradigm LOCATE expects that the segments identified in an OUTPUT directive are destined for read-only memory yet the segment seg in class *'class'* is assigned to a region mapped as *memtype*. While this condition is inappropriate for ROM-based execution (the segment won't be available if not in EPROM), it is permitted for downloading a segment to RAM and the warning can be ignored.

W1033 Class *'class'* not named in an OUTPUT directive
The named class is in a region of the memory address space defined with the read-only attribute but the class was not named in a configuration file OUTPUT directive. This warning may indicate a potential problem since the class would not be in an EPROM if the class is not part of an OUTPUT directive.

W1034 All segments have been removed from class *'class'*
All of the segments in the named class have been assigned addresses using the SEGMENT directive. Including the class in an ORDER directive has no effect on the address assignments and can be eliminated.

W1035 Debug information nesting error, fixup applied
Paradigm LOCATE has detected a scoping error in the input debug information and has attempted to fix the error by supplying the missing scopes. This warning is usually accompanied by a warning from the compiler that debug information was lost due to the complexity of the input source file. Fix the problem in the source module to get rid of this warning.

W1036 Lack of debug information prevents structure padding
This warning occurs when the debug information is insufficient or does not accurately indicate the size of a structure member. You can use the **-d2** option to identify which module is responsible for the faulty debug information.

W1037 **Ambiguous debug information, translation not possible**

Assembly language modules with absolute segments are usually the culprit.

The input debug information is incomplete and Paradigm LOCATE is unable to completely translate it.

W1038 **Can't translate register variable using two registers**

The input debug information contains register variable pairs not supported by Paradigm DEBUG and the debug information is lost.

W1039 **Segment *'seg/class'* has been truncated in file *'file'***

This warning is output by the evaluation version of Paradigm LOCATE when a segment exceeds the internally set limits. Because the segment has been arbitrarily truncated, the application may no longer work correctly although the debugging information attached is still intact.

W1040 **TRUNCATE option ignored in *'file'*, line *'nnn'***

The TRUNCATE option can only be used with binary files.

W1041 **'option' option in *'file'*, line *'nnn'* is obsolete**

The named option is no longer supported by Paradigm LOCATE and has been replaced with improved capabilities.

W1042 **Listing file can't process case insensitive links**

Paradigm LOCATE requires that case-insensitive symbols be used in order to demangle C++ names in the listing file.

W1043 **'option' option in *'file'*, line *'nnn'* is not supported**

The named option is not supported by this version of Paradigm LOCATE.

W1044 **Bad CodeView debug information, fixup applied - *'nnn'***

The CodeView debugging information on the input load module (the .ROM or .EXE file created by the linker) was found to be corrupt. Paradigm LOCATE has done its best to work around the problem but some debugging information may be lost.

W1045 **Bad Borland debug information, fixup applied - *'nnn'***
The Borland debugging information on the input load module (the .ROM or .EXE file created by the linker) was found to be corrupt. Paradigm LOCATE has done its best to work around the problem but some debugging information may be lost.

W1046 **'type' checksum skipped for *'segment'/'class'***
The named segment is not declared in an OUTPUT directive yet appears in a checksum calculation. Paradigm LOCATE will only calculate checksums on segments identified in OUTPUT directives.

W1047 **Unable to fixup virtual segment *'seg'* at *'seg:off'***
The specified segment fixup in the relocation table could not be translated. This error usually indicates the load module and segment map were not created on the same linker run, or the input files are corrupt.

W1048 **Mismatch in load module size and segment map size**
The size of the load module and the segment map don't agree in size. This may or may not be a problem but you can get rid of this warning by completely defining all segments in the load module by avoiding the use of DUP '*nnn'* (?) constructs in your code.

W1049 **C++ namespaces present - Paradigm DEBUG 6.0 or later recommended**
The debug information contains namespace information but the output is for an earlier version of Paradigm DEBUG that lacks namespace support.

Preprocessor warnings

Message explanations

The following warning diagnostics are produced by configuration file preprocessor during the parsing of the configuration file.

W2000 **Macro *'macro'* needs argument in *'file'*, line *'nnn'***
An argument was expected with the macro.

W2001 **Wrong number of arguments 'args' in *'file'*, line *'nnn'***
The wrong number of macro arguments was detected during macro expansion.

W2002 **Expected formal parameter in *'file'*, line *'nnn'***
A formal parameter was expected by Paradigm LOCATE.

W2003 **Undefined symbol *'symbol'* in expression in *'file'*, line *'nnn'***
A symbol that has not been defined in a configuration or on the command line was used in an expression.

A P P E N D I X

B

Error diagnostics

The errors listed in this appendix indicate the existence of a serious problem that prevents Paradigm LOCATE from producing useful output. Each of the error messages are listed in numerical order for easy lookup.

Paradigm LOCATE errors

Message explanations

The following error diagnostics are produced by Paradigm LOCATE while the processing the input files, command line arguments, or configuration file.

E1000 **Internal error *'id'* - contact Paradigm Systems**
A serious internal error has been detected by Paradigm LOCATE. Please contact Paradigm Systems with the internal error ID for assistance in resolving the error.

E1001 **Error opening *'file'* - *'err_info'***
Paradigm LOCATE was unable to open the specified file for the reason given in *err_info*.

E1002 Error reading *'file' - 'err_info'*

Paradigm LOCATE was unable to satisfy a read of the named file for the displayed reason. This error usually indicates an incomplete load module or some other serious error.

E1003 Error writing *'file' - 'err_info'*

Paradigm LOCATE was unable to complete a write to file for the reason *err_info*. The most likely cause of this error is a device with no space - a full disk.

E1004 Insufficient memory available for Paradigm LOCATE

The dynamic memory requirements needed by Paradigm LOCATE are unavailable to complete the processing. Attempt to free up some memory and retry the operation or reduce the amount of debug information in the load module if this error is encountered.

E1005 Unable to find configuration file *'file'*

The Paradigm LOCATE configuration file *'file'* could not be found. Check that the configuration file exists in the directory with the relocatable load module or in the directory specified by the **-c** command line option. If the **-c** command line option is not used, Paradigm LOCATE assumes that the configuration file has the same name as the relocatable load module with a .CFG extension and that it is located in the same directory as the relocatable load module (.ROM file), for example, `locate -cdemo.cfg demo.rom`.

E1006 Paradigm LOCATE input/output filenames must be unique

To avoid confusion and preserve all files, Paradigm LOCATE does not permit the input and output filenames to be the same. This error will most likely occur when the output file extension is .EXE and the input file also has the .EXE file extension. The workaround is to have the linker name the output file .ROM (relocatable load module) or some other extension of your choosing.

E1008 Unable to fixup virtual segment *'seg'*

This is usually caused by absolute segments.

The specified segment fixup in the debug information could not be converted to an absolute segment address.

E1009 Unable to fixup program entry point - *'seg:off'*
The program entry point failed segment translation. Since the entry point must be in a defined segment, this error is likely to be accompanied by a more serious error. Often this error is caused by trying to process an input file that was packed by the Microsoft linker.

E1010 Unable to fixup initial stack - *'seg:off'*
The program stack failed segment translation. Since the stack initialization is picked up from the segment with the stack attribute, this error is likely due to the lack of a stack segment in the application. Often this error is caused by trying to process an input file that was packed by the Microsoft linker.

E1011 New executable file *'file'* is not supported
Paradigm LOCATE does not support new style (Microsoft Windows or OS/2) executable files. Check your linker options and select the original DOS .EXE file format.

E1012 Corrupted relocatable load module in file *'file'*
Paradigm LOCATE has determined the header on the load module is corrupt or the file is not in the EXE format. Check your Paradigm LOCATE command line options. Be sure that you pass the .ROM or .EXE as an input file, for example, `locate demo.rom -cdemo.cfg`.

E1013 Input file *'file'* is already an AXE file
The named file is already in AXE format, most likely because the file has been processed by Paradigm LOCATE.

E1014 Multiple segment fixup records detected in *'file'*
Only one segment fixup for a single location is allowed. Should this error occur, contact Paradigm Systems for assistance.

E1015 Size must be between 1 and 1024 in *'file'*, line *'nnn'*
The EPROM size specified in the HEXFILE SIZE option must be an integer between 1 and 1024. Note that the size value is in KB, for example, size=8 means 8096 bytes.

E1016 Fill argument must be between 0 and 255 in *'file'*, line *'nnn'*
The EPROM fill character specified in the HEXFILE FILL option must be in the range 0x00 to 0xFF.

E1017 Offset must be in 1MB address space in *'file'*, line *'nnn'*
The EPROM offset specified in the HEXFILE OFFSET option must be in the range 0x00000 to 0xFFFFF.

E1018 Split argument must be 1, 2 or 4 in *'file'*, line *'nnn'*
The EPROM split specified in the HEXFILE SPLIT option must be either 1 for no split, 2 for a pair of EPROMs or 4 if a 32-bit wide split is required.

E1019 Unable to split Intel extended hex in *'file'*, line *'nnn'*
Paradigm LOCATE does not split Intel extended hex files. If your design requires a set of EPROMs, the Intel hex, binary or Tektronix hex output formats must be used.

E1020 Length must be between 8 and 64 bytes in *'file'*, line *'nnn'*
The HEXFILE LENGTH option accepts a hex file record length of 8 to 64 bytes in length.

E1021 Unable to find segment map in *'file'*
Paradigm LOCATE is unable to find the segment map in the linker map *file*. The segment map is needed by Paradigm LOCATE to find and extract the individual segments from the relocatable load module.

E1022 Syntax error at or near *'this'* in *'file'*, line *'nnn'*
The syntax of the specified configuration file directive is in error and must be corrected. Note that the line number used to identify the error may be after the point of the error if the line has been continued one or more times.

E1023 Unknown class *'class'* in *'file'*, line *'nnn'*
Paradigm LOCATE is unable to find the *class* named class in the list of classes. Make sure that the class name is spelled exactly as it appears in the linker map (.MAP) since Paradigm LOCATE uses case-sensitive class names.

E1024 **Unknown segment *'seg'* in *'file'*, line *'nnn'***
Paradigm LOCATE is unable to find the segment named *seg* in the list of segments. Make sure that the segment name is spelled exactly as it appears in the linker map (.MAP) since Paradigm LOCATE uses case-sensitive segment names.

E1025 **Missing or unsupported CPU type in *'file'*, line *'nnn'***
The target microprocessor field in the CPUTYPE directive is either unsupported, missing or multiply defined.

E1026 **CPU does not support the initialization in *'file'*, line 'nnn'**
The target microprocessor specified in the CPUTYPE directive cannot perform the identified peripheral register initialization. Either change the target microprocessor defined in the CPUTYPE directive or use the generic port I/O options of the INITCODE directive.

E1027 **I/O port address too large in *'file'*, line *'nnn'***
The I/O port address must be in the range of 0x0000 to 0xFFFF.

E1028 **One or more classes required in *'file'*, line *'nnn'***
The specified directive requires at least one class to be named in the list of classes.

E1029 **Two or more classes required in *'file'*, line *'nnn'***
The specified directive requires two or more classes to be named in the list of classes.

E1030 **Illegal warning control option in *'file'*, line *'nnn'***
One or more of the warnings specified in the WARNINGS directive do not correspond to a valid warning ID.

E1031 **MAP directive address range error in *'file'*, line *'nnn'***
A valid region requires that the first address in a MAP directive be less than or equal to the second address.

E1032 **Class *'class'* must be DUPLICATEd before compression**
It is not possible for a class to decompress on to itself.
You are attempting to compress a class that has not been duplicated or does not have a zero-length segment as the first segment in the class.

E1033 Compressed class *'class'* too large during pass 2
Paradigm LOCATE runs a two pass compression algorithm, the first pass to estimate the size of the compressed class, which is needed to apply segment fixups. A second pass is then performed, after segment fixups have been applied, to compress the class. On pass 2, the class compressed less than expected, generating this error.

E1034 Unknown or illegal command line option *'option'*
The specified command line option is incorrect and requires fixing before Paradigm LOCATE will continue.

E1035 SPLIT option incompatible with Intel extended hex
The command line option to split the EPROM files is incompatible with Intel extended hex output. If your design requires a set of EPROMs, the Intel hex, binary or Tektronix hex output formats must be used.

E1036 SIZE argument out of range in option *'option'*
The EPROM size specified in the **-Hd** command line option must be a power of 2. Valid EPROM sizes (in KB) are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024.

E1037 OFFSET argument out of range in option *'option'*
The offset field in the **-Ho** command line option must be a 20-bit unsigned integer.

E1038 FILL argument out of range in option *'option'*
The EPROM fill character specified in the **-Hf** option must be in the range 0x00 to 0xFF.

E1039 SPLIT argument out of range in option *'option'*
The EPROM split specified in the **-Hs** command line option must be either 1 for no split, 2 for a pair of EPROMs or 4 if a four EPROM set is required.

E1040 LENGTH argument out of range in option *'option'*
The hex record length specified in the **-Hl** command line option must be between 8 and 64.

E1041 Diagnostics level out of range in option *'option'*

The diagnostics output level specified in the **-d** command line option must be either 0 for no diagnostics, 1 for filename diagnostics, 2 for filename and module diagnostics, 3 for compression statistics, or 4 to enable all diagnostics.

E1042 Illegal or out of range warning argument in option *'option'*

The warning ID in the **-w** command line option is not a valid warning ID.

E1043 Debug information version is not supported

The debug information supplied to Paradigm LOCATE is beyond the currently supported version. This error is most likely due to a compiler or linker update by the compiler vendor.

E1044 Packed CodeView debugging information not supported

The Microsoft CVPACK utility was used to pack the debugging information, preventing Paradigm LOCATE from processing the file.

E1045 Unpacked CodeView debugging information not supported

CVPACK usually fails to run when there is a linker error.

Paradigm LOCATE expects to see packed debug information, so something prevented CVPACK from successfully completing.

E1046 Bad or missing CV2 debug information - *'code'*

An error occurred translating the Microsoft CodeView debug information. Indication of corrupted debug information found in the .ROM or .EXE file that was created by the linker. Please contact Paradigm Systems with the details of this error. Uploading your application (.ROM, .MAP and configuration file) will help our technical support group resolve this problem more quickly.

E1047 Bad or missing CV4 debug information - *'code'*

An error occurred translating the Microsoft CodeView debug information. Indication of corrupted debug information found in the .ROM or .EXE file that was created by the linker. Please contact Paradigm Systems with the details of this error. Uploading your application (.ROM, .MAP and configuration file) will help our technical support group resolve this problem more quickly.

E1048 Bad or missing Borland TD2 debug information - *'code'*

An error occurred translating the Borland debug information. Indication of corrupted debug information found in the .ROM or .EXE file that was created by the linker. Please contact Paradigm Systems with the details of this error. Uploading your application (.ROM, .MAP and configuration file) will help our technical support group resolve this problem more quickly.

E1049 Bad or missing Borland TD3 debug information - *'code'*

An error occurred translating the Borland debug information. Indication of corrupted debug information found in the .ROM or .EXE file that was created by the linker. Please contact Paradigm Systems with the details of this error. Uploading your application (.ROM, .MAP and configuration file) will help our technical support group resolve this problem more quickly.

E1050 Bad or missing Borland TD4 debug information - *'code'*

An error occurred translating the Borland debug information. Indication of corrupted debug information found in the .ROM or .EXE file that was created by the linker. Please contact Paradigm Systems with the details of this error. Uploading your application (.ROM, .MAP and configuration file) will help our technical support group resolve this problem more quickly.

E1051 'name' debug information exceeds translation limits

The named debug records exceeds the capacity of the output file format. The only solution is to eliminate some modules with debug information and re-run Paradigm LOCATE.

E1052 CHECKSUM directive address range error in *'file'*, line *'nnn'*

A valid checksum region requires that the first address in a CHECKSUM directive be less than or equal to the second address.

E1053 CHECKSUM FILL option out of range error in *'file'*, line *'nnn'*

The fill character specified in the CHECKSUM directive must be in the range 0x00 to 0xFF.

E1054 **CHECKSUM type option not specified in *'file'*, line *'nnn'***
The CHECKSUM record type is incorrectly specified. Please select one of following CHECKSUM type options: ROMBIOS, CRC16, or CRC32.

E1055 **ADDRESS option out of range error in 'file', line 'nnn'**
The ADDRESS option in the CHECKSUM directive is outside the target system memory address space.

E1056 **ADDRESS cannot be part of checksum in *'file'*, line *'nnn'***
You can not specify an address to place the checksum that is inside the range of the checksum calculation.

E1057 **Include file size cannot be greater than 64KB in *'file'*, line *'nnn'***
The binary include file size has exceeded the 64KB limit.

E1058 **DUP must copy class 'class' to a unique class in *'file'*, line *'nnn'***
You cannot duplicate a class to itself.

Preprocessor errors

Message explanations

The following error diagnostics are produced by configuration file preprocessor during the parsing of the configuration file.

E2000 **Internal error *'num'* - contact Paradigm Systems**
This is issued by all preprocessor internal errors. Please contact Paradigm Systems should you encounter an internal error.

E2001 **Conditional block nesting error in *'file'***
Your configuration file has incorrectly nested **#if/#else/#endif** directives.

E2002 Conditional without an argument in *'file'*, line *'nnn'*
You used a conditional directive but failed to provide an expression to evaluate.

E2003 #include syntax error in *'file'*, line *'nnn'*
#include requires the name of the include file enclosed in either double quotes (") or left ("<") and right angle (">") brackets.

E2004 #else may not follow #else in *'file'*, line *'nnn'*
An #else clause can only follow an **#if** or **#elif** directive.

E2005 #endif must be in an #if block in *'file'*, line *'nnn'*
Paradigm LOCATE found an **#endif** with a corresponding **#if** directive.

E2006 Unsupported #control definition in *'file'*, line *'nnn'*
An unsupported preprocessor control was found. Valid controls are **#if**, **#else**, **#endif**, **#elif**, **#define**, **#undef**, and **#include**.

E2007 Include file '*incfile*' not found in *'file'*, line *'nnn'*
The named include file could not be found. Check that the path specifies the correct location of the file.

E2008 Too many nested *'token'* statements in *'file'*, line *'nnn'*
You broke the preprocessor with a configuration file beyond comprehension. You are going to have to simplify the file before continuing.

E2009 Macro expansion error in *'file'*, line *'nnn'*
An error occurred when expanding a macro. Identify the macro in error and correct the problem.

E2010 Redefining defining variable *'var'* in *'file'*, line *'nnn'*
Another **#define** for the same variable has been found. Use the **#undef** directive before redefining the variable.

E2011 #define syntax error in *'file'*, line *'nnn'*
You must specify a variable name for the macro you wish to define.

E2012 **Illegal #undef argument in *'file'*, line *'nnn'***
#undef requires that a macro name be supplied.

E2013 **End of file in macro argument in *'file'*, line *'nnn'***
An end of file condition was found while processing the macro argument list. Check the macro and correct before continuing. This error can also occur if the end of the file is reached while processing a C comment.

E2014 **Recursive macro definition *'macro'* in *'file'*, line *'nnn'***
Recursive macros are not permitted. Correct the error before continuing.

E2015 **Empty character constant in *'file'*, line *'nnn'***
A character constant was expected but not found.

E2016 **Unterminated string or character constant in *'file'*, line *'nnn'***
An improperly terminated string literal or character constant was found.

E2017 **Can't use string in #if in *'file'*, line *'nnn'***
String literals are not valid in conditional expressions.

E2018 **Bad #if defined in *'file'*, line *'nnn'***
An expression that could not be evaluated was found.

E2019 **Assignment not allowed in #if in *'file'*, line *'nnn'***
Use of the assignment operator is not permitted in conditional expressions.

E2020 **Error in multiline #if in *'file'*, line *'nnn'***
The multiline **#if** directive needs work before it can be accepted by Paradigm LOCATE.

E2021 **Divide by zero error in *'file'*, line *'nnn'***
The result of an expression evaluation resulted in division by zero.

E2022 **#if stack overflow in *'file'*, line *'nnn'***
Too many nested **#if** directives has been found, you will have to simplify the configuration file.

E2023 **Operator *'op'* context fault in *'file'*, line *'nnn'***
This is an inappropriate use of the named operator.

E2024 **Expression error in *'file'*, line *'nnn'***
Paradigm LOCATE was unable to evaluate the expression. Correct or simplify before continuing.

E2025 **#define syntax error in command line option *'opt'***
A macro defined with the **-D** command line option is incorrectly formed.

E2026 **#error in *'file'*, line *'nnn'*: 'errmsg'**
A **#error** directive in your configuration was processed.

E2027 **Macro exceeds preprocessor limit in *'file'*, line *'nnn'***
A macro definition may have been too long and needs to be simplified and shortened.

APPENDIX C

Exit codes

The exit code returned by Paradigm LOCATE can be used by MAKE utilities or batch files to determine the success or failure of the processing. The following table indicates the meaning assigned to each error code.

Table C.1
LOCATE exit codes

Exit Code	Meaning
0	No errors, possibly warnings
1	Error(s)
2	Serious error
3	Critical or fatal error

The severity of errors depends on the action which caused the error. Regular errors are unexpected conditions detected with the conversion of relocatable input file to an absolute output file, including the conversion of type information. Some errors terminate processing immediately while others continue until other exceptional conditions have been checked.

Serious or critical errors are associated with the operating system of I/O operations and cause Paradigm LOCATE to immediately finish, clean up and exit.

The WARNINGS EXITCODE option can also be used to set the exit code for warning conditions.

Paradigm LOCATE has the **-W** option to generate a non-zero exit code should any warnings be detected during processing. This option should be used when an environment might not display any messages and an indication of warning is required.

APPENDIX D

INITCODE port definitions

The Paradigm LOCATE INITCODE directive can be used to initialize peripheral registers found in the Intel 80C186 and NEC V-Series microprocessors. This capability is especially attractive since it permits memory and peripheral chip selects, wait states, and DRAM refresh devices to be initialized before the application startup code takes control of the CPU, without the need to modify the startup code. By avoiding the need to customize the startup code with complex segmentation and initialization code, the user can focus on more interesting applications.

Only peripheral devices which impact memory initialization are supported.

Table D.1 uses the standard peripheral register names as defined by each microprocessor vendor. The table is ordered by microprocessor, as it is used in the CPUTYPE directive. If a specific microprocessor does not appear in the following table, it does not support any port initializations.

Table D.1 INITCODE port definitions

CPUTYPE	Register	Port address
I80186	UMCS	FFA0H
I80188	LMCS	FFA2H
	PACS	FFA4H
	MMCS	FFA6H
	MPCS	FFA8H

CPUTYPE	Register	Port address
I80C186	UMCS	FFA0H
I80C186XL	LMCS	FFA2H
I80C188	PACS	FFA4H
I80C188XL	MMCS	FFA6H
	MPCS	FFA8H
	MDRAM	FFE0H
	CDRAM	FFE2H
	EDRAM	FFE4H
I80C186EA	UMCS	FFA0H
I80C188EA	LMCS	FFA2H
I80L186EA	PACS	FFA4H
I80L188EA	MMCS	FFA6H
	MPCS	FFA8H
	RFBASE	FFE0H
	RFTIME	FFE2H
	RFCON	FFE4H
I80C186EB	GCS?ST	FF80H-FF9EH
I80C188EB	GCS?SP	FF80H-FF9EH
I80L186EB	LCSST	FFA0H
I80L188EB	LCSSP	FFA2H
	UCSST	FFA4H
	UCSSP	FFA6H
	RFBASE	FFB0H
	RFTIME	FFB2H
	RFCON	FFB4H
I80C186EC	GCS?ST	FF80H-FF9EH
I80C188EC	GCS?SP	FF80H-FF9EH
	LCSST	FFA0H
	LCSSP	FFA2H
	UCSST	FFA4H
	UCSSP	FFA6H

CPUTYPE	Register	Port address
I80C186EC/188EC continued	RFBASE	FFB0H
	RFTIME	FFB2H
	RFCON	FFB4H
	MPICP0	FF00H
	MPICP1	FF02H
	SPICP0	FF04H
	SPICP1	FF06H
AM186ED AM186EM/188EM AM186ER/188ER AM186ES/188ES	UMCS	FFA0H
	LMCS	FFA2H
	PACS	FFA4H
	MMCS	FFA6H
	MPCS	FFA8H
	IMCS	FFA0H **(ER only)**
	PDCON	FFF0H **(EM ER only)**
	PIOMODE0	FF70H
	PIODIR0	FF72H
	PIOMODE1	FF76H
	PIODIR1	FF78H
	MDRAM	FFE0H **(EM ER ES only)**
	CDRAM	FFE2H
	EDRAM	FFE4H
	SYSCON	FFF0H **(ES ED only)**
	AUXCON	FFF2H **(ES ED only)**
	WDTCON	FFE6H **(ES ED only)**

CPUTYPE	Register	Port address
AM186CC	UMCS	FFA0H
	LMCS	FFA2H
	PACS	FFA4H
	MMCS	FFA6H
	MPCS	FFA8H
	PIOMODE0	FFC0H
	PIODIR0	FFC2H
	PIOMODE1	FFCAH
	PIODIR1	FFCCH
	PIOMODE2	FFD4H
	PIODIR2	FFD6H
	CDRAM	FFAAH
	EDRAM	FFACH
	WDTCON	FFE0H
	SYSCON	FFF0H
I80386EX	CS?ADL	F400H-F436H
	CS?ADH	F400H-F436H
	CS?MSKL	F400H-F436H
	CS?MSKH	F400H-F436H
	UCSADL	F438H
	UCSADH	F43AH
	UCSMSKL	F43CH
	UCSMSKH	F43EH
	RFSBAD	F4A0H
	RFSCIR	F4A2H
	RFSCON	F4A4H
	RFSADD	F4A6H
	ICW1M	F020H
	ICW2M	F021H

CPUTYPE	Register	Port address
I80386EX continued	ICW1S	F0A0H
	ICW2S	F0A1H
	P1CFG	F820
	P2CFG	F822H
	P3CFG	F824H
	PINCFG	F826H
	REMAPCFG	0022H
D70208 V40	RFC	FFF2H
D70216 V50	WMB	FFF4H
	WCY1	FFF5H
	WCY2	FFF6H
D70208H V40H	RFC	FFF2H
D70216H V50H	SCTL	FFF7H
	WMB	FFF4H
	WCY1	FFF5H
	WCY2	FFF6H
	WCY3	FFEAH
	EXMB	FFEDH
	WSMB	FFECH
	WIOB	FFEBH
D70320 V25	IDB	[IDB]00:0FFFH
D70325 V25+	RFM	[IDB]00:0FE1H
D70330 V35	WTC	[IDB]00:0FE8H
D70335 V35+	PRC	[IDB]00:0FEBH
	PMC0	[IDB]00:0F02H
	PMC1	[IDB]00:0F0A
	PMC2	[IDB]00:0F12

CPUTYPE	Register	Port address
D70236 V53	RFC	FFF2H
	WMB0	FFEAH
	WMB1	FFF3H
	WCY0	FFECH
	WCY1	FFEBH
	WCY2	FFF4H
	WCY3	FFF5H
	WCY4	FFF6H
	WAC	FFEDH
	SBCR	FFF1H
D70423 V55SC **D70433 V55PI**	PRC	FFFEFH
	RFM	FFFECH
	MBC	FFFEAH
	PWC0	FFFE8H
	PWC1	FFFE9H

APPENDIX E

AXE utility

The AXE utility is a program which displays various statistics about AXE86 files created by Paradigm LOCATE. The fields displayed from the input AXE file are

- program entry point
- AXE header size
- region list
- segment list

The AXE utility first looks for a file extension of .AXE before trying to open a file with the .EXE extension. The format of the AXE command line is

```
axe filename[.ext]
```

The following figure contains sample output from the AXE utility, together with a brief description of each section in the AXE file header.

Figure E.1
AXE header information

```
1)    AXE Version 1.00
      Entry Point: FFFF:0000
      AXE Header Size: 256 bytes

2)    \LOCATE\DEMO\SIEVE.AXE contains 3 regions
      000000  03FFFF  Read/Write
      040000  0F7FFF  No access
      0F8000  0FFFFF  Read Only

3)    \LOCATE\DEMO\SIEVE.AXE contains 6 segments
```

```
    0  F800:0000 00133  O-   000100
    1  F813:0004 00010  O-   000280
    2  F815:0000 00004  O-   000300
    3  F816:0000 00010  O-   000380
    4  FFF0:0000 00013  O-   000400
    5  FFFF:0000 00005  O-   000480

Load module size: 367 bytes
```

Section 1 is the AXE header information, containing the version of AXE file, the program entry point, and the size of the AXE segment descriptor buffer.

Section 2 is the region map, displaying the mapping instruction for the target system memory from the Paradigm LOCATE MAP directives. The first item is the starting address of the region, the second address is the ending address of the region followed by the access type of the region.

The load module size is the sum of the sizes of each segment in the AXE file.

The segment map in section 3 lists the segment index, the segment base address and segment length, segment attributes, and the offset of the segment within the AXE file. The first segment attribute indicates whether the segment is read-write ('-') or if it is read-only ('O'). The second attribute indicates whether the segment is present in the AXE file ('-') or if the segment descriptor is provided as a reference ('R') but the segment doesn't actually exist.

A P P E N D I X

F

Hex file formats

This appendix documents the Intel hex file formats used by Paradigm LOCATE. This information is provided to those users that need to read Intel hex or extended hex file formats created by the Paradigm LOCATE HEXFILE configuration file directive.

Intel extended hex

Intel extended hex is a file format designed to represent binary data within the 80186-family address space using the standard ASCII character set. The hexadecimal representation of each binary byte is encoded in a pair of ASCII characters in the range '0' - '9' and 'A' to 'F'.

There are four different record types which make up the Intel extended hex file format:

- Extended Address Record
- Start Address Record
- Data Record
- End of File Record

Each Intel extended hex record begins with a colon (':') character as the record mark. The *record mark* field is then followed by a *record*

length field which specifies the number of bytes of information that follow the *record type* field.

Each record ends with a *checksum* field that contains the ASCII representation of the two's complement of the binary data from the record length field. If the record is correct, the sum of all fields, including the checksum field, will be zero.

Extended Address Record

The Extended Address record is used to define a *segment base address* (SBA) for the following Data records, which supply the offsets for each data record from this base address.

The segment base address is zero until it is defined in an Extended Address record. Once defined, the segment base address will remain in effect until a subsequent Extended Address record is encountered.

Mark	Length	Offset	Type	SBA	Checksum
':'	'02'	'0000'	'02'	'XXXX'	'XX'

Data Record

Each Data record defines part of the memory address space of the application. The absolute address of a Data record is determined by the Offset field and the segment base address in the last Extended Address record.

Mark	Length	Offset	Type	Data	Checksum
':'	'XX'	'XXXX'	'00'	'XXXXXXXX'	'XX'

The Length field is determined by the amount of data to be output and the LENGTH option in the configuration file HEXFILE directive.

Start Address Record

The Start Address record is used to specify the program entry point for the application, as computed by Paradigm LOCATE.

Mark	Length	Offset	Type	CS	IP	Checksum
':'	'04'	'0000'	'03'	'XXXX'	'XXXX'	'XX'

Paradigm LOCATE will always set this record to the program entry point, enabling Intel extended hex file loaders to automatically set CS:IP to the first instruction of the application.

End of File Record

This record marks the end of the Intel extended hex file and is always the last record output by Paradigm LOCATE.

Mark	Length	Offset	Type	Checksum
':'	'00'	'0000'	'01'	'FF'

Intel hex

This is the original Intel hex file format, dating back to the days of the 8080 microprocessor. Being the original hex file format for Intel microprocessors having a 64KB address space, the Intel extended hex file format added the Start Address and Extended Address record types to expand the address space to the 1MB used in the 8086/88 and subsequent 16-bit microprocessors.

Intel hex file is often used with 16-bit data paths since Intel extended hex can't be represented in a split format because of the Extended Address records. Still, Intel hex has its limitations since it can never support more than 64KB of data per file.

Tektronix hex

Tektronix hex, also referred to as Tekhex, is also a file format designed to represent of top 64KB of binary data using the standard ASCII character set. The hexadecimal representation of each binary byte is encoded in a pair of ASCII characters in the range '0' - '9' and 'A' to 'F'.

Each Tekhex record begins with a slash ('/') character as the record mark. The *record mark* field is then followed by the *load address* and a *record length* fields which specify the offset and count of the data that follow.

Both the header and data fields have a *checksum* field that contains sum, modulo 256, of the data in the preceding records.

Data Record

Each Data record defines part of the memory address space of the application. The absolute address of a Data record is determined by the Offset field.

Mark	Offset	Length	Chk1	Data	Chk2
'/'	'XXXX'	'XX'	'XX'	'XXXXXXXX'	'XX'

The Length field is determined by the amount of data to be output and the LENGTH option in the configuration file HEXFILE directive.

I N D E X

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X