USLink®

Embedded Linker/Locator

User's Guide

Revision 2.15
December 1999

U S SOFTWARE®
EMBEDDED EXCELLENCE

www.ussw.com

Copyright and Trademark Information

United States Software Corporation
7175 NW Evergreen Parkway, Suite 100
Hillsboro, OR 97124
(503) 844-6614
Fax (503) 844-6480
E-mail: support@ussw.com

Quick Contents

Documentation Conventions

Computer output and code examples: Courier, usually in a separate paragraph.

Function names and command names: ***Bold italic***, usually followed by parentheses, as in ***main()*** function.

Variables: Courier 11 italic (the variable *filename*).

File names: Times bold (the file **usrclk.asm**), in lower case.

Key names: Initial capital, in angle brackets, as in press <Enter>.

Menu names and selections, dialog box names, screen titles, window titles: Times bold, as in **File** menu**.**

Notes: Indicate important information.

Cautions: Indicate potential damage to hardware or data.

Warnings: Indicate potential injury to users.

Revision History

Revision Number	History	Date
1.0	Original issue	12/96
1.1	Reorganized and reformatted	11/97
2.15	Revised content	12/99

Contents

1. Introduction

1

Chapter Contents

Manual Organization

The *USLink® User's Guide* contains the following chapters:

1. Introduction
Describes the location process, explains features of and how to use the linker, and gives troubleshooting guidance.

2. 32-bit Protected-Mode Applications
Look in this chapter to find a worked-out example of locating in ROM (and RAM) a 32-bit protected-mode application.

3. Command Reference
The linker **.cmd** command file helps make locating applications easy and fast. This chapter describes command files in detail and also provides a detailed explanation of each of the commands that can go into them.

A. Error and Warning Messages
Explains the error and warning messages that the linker generates.

Linker Overview

The LINKER links and locates real-mode, protected-mode, and mixed-mode applications by performing commands that you give to it in a command file.

Since many programmers are new to the location process, we provide a general description of this important process starting on the next page. Then we provide an overview of the capabilities of the linker in the next section of this chapter, highlighting its distinctive features. This is followed by a general discussion of the input the linker requires and the output it produces. A map file is a sort of listing file always generated by the linker and that file type is discussed next. The next topic is toolchains that the linker supports. The final section of this chapter provides troubleshooting assistance.

The Location Process

Linking/Locating

The simple story of the linking-locating process is that linkers order and link program segments together, while locators assign addresses to them. Linkers in fact do more than that and locators similarly have historically taken on more duties than just address assignment. We will assume that you have a general familiarity with what goes on in the linking process, but we will take a closer look at the location process here.

Native versus Embedded Development

Native-development environments today shield the user almost entirely from the locating process because native applications are practically always relocatable. Embedded-system developers, however, need to know about locating because their applications must usually be absolutely located. The primary difference between relocatable and absolutely located applications is that the latter have fixed addresses.

Location Units

When building an absolutely located application, you work with the following three basic structures:

Segments Protected-mode 32-bit segments can be as large as 4GB. Of the three possible 80x86-family type of segments (namely data, code, and system), applications programmers create and work primarily with the data and

code segments. The embedded linker can be used to set up system segments (e.g., those containing tables for the `GDT, IDT, LDT, TSS,` and the various trap and interrupt gates), as well as to define code and data segments at location time that don't exist until then, but are used by the user's application.

Classes Classes are collections of segments, either grouped together according to user specifications or classed together by the linker to organize modules. For example, all of the segments in one class might contain initialized data, and all the segments in another class might contain initialization code or code written in assembly language.

Groups Groups are also collections of segments, but each of the segments in a group has the same segment base. So, the entire group must be within the segment-size limit defined by the processor mode, which is 4GB for 32-bit protected mode.

Know Where Your RAM and ROM Are

Consider the following when locating these structures in memory:

- You need to know the starting addresses and sizes of RAM and ROM chips on your target board.
- Segments that don't have load-image data (for example, the Stack and BSS segments) can be located, but their contents will be undefined because no load-image data is present. If you use ***LOCATE*** to specify that the data images of such segments be written to the target, those data images will be set to zero. Also, zero-length segments are not placed in the output **.abs**, **.hex**, or **.bin** files.

Default Location Order

All segments not explicitly located using commands are located in the order the segments are defined in object and library files, or the order in which they are created with the linker command ***CREATE***.

Typically you will want to locate code in ROM and data in RAM. The examples worked out in Chapter 3 should help you get a feel for using the linker and for the location process generally.

Features

1

What the Linker Does

The linker links object and library files, then orders and absolutely locates in memory the segments, classes, and groups that constitute the 80x86-family (aka *Intel Architecture*), creating code that can be loaded into RAM using a debugger or burned into ROM to create an embedded application.

Supported Compilers

The linker supports applications built with toolchains (that is, assemblers and C/C++ compilers) from the following vendors:

- Borland
- MetaWare
- Microsoft
- Watcom

Protected-Mode Structures

The linker can construct 32-bit protected-mode CPU structures, including the `GDT, IDT, LDT`, gates, page tables, and `TSS`s, and it supports multiple-mode or mixed-mode applications.

Initialized RAM Data

If your application has initialized data that you want to reside in RAM, the linker can pack the segments containing it for storage into ROM, and then at boot-up time your application's startup code can unpack the data and copy it into RAM by using a macro supplied with the linker. This unpack-and-copy macro is named ***raminit_32p*** (32-bit protected), and can be found in the file **link.inc.**

Other Support Macros

Here is a list of other support macros (and what they do) that can also be found in the file **link.inc**:

def_alias	Defines alias segments.
def_gate	Defines CALL/TASK/TRAP/Interrupt gates.
def_noentrygate	Defines gate without default entry point.
def_init	Defines RAM `init` table segment.
def_tbl	Defines `GDT, IDT, LDT`, and page tables.
def_tss	Defines `TSS` segments.
sup_defseg	Supports ***def_xxx*** macros.
sup_init32	Supports ***raminit_xxx*** macros.

Input and Output

1

Possible Output Formats

To make the linker easy to use, all commands and options are read from a sequential command file (see the first two sections of Chapter 3 for a detailed discussion of the **.cmd** command file), and the user controls the ultimate output format.

Possible output formats of your located application are:

- Intel OMF86 absolute
- Intel OMF286 absolute
- Intel OMF386 absolute
- Intel extended 86 hex
- Intel 32-bit hex
- Binary

Default extensions for output files are:

absolute	**.abs**
hex	**.hex**
binary	**.bin**

A type of listing file, called a map file (with **.map** extension), is always generated by the linker; a example annotated **.map** file begins on the next page.

Example Map File

Command file listing:

```
_____________________________________

Processing command file:
_____________________________________

[1] object csistart.obj
[2] object cmain.obj cutils.obj io.obj
[3] cpu 386ex
[4] abs386
[5] map csamp.map
[6] flat
[7] debug
[8] create GDTSYS
[9] create IDTSYS
[10] create tss386_sys
[11] create raminit
[12] locate group FLAT_DATA segment _STACK :: 40000P
[13] locate init_text :: 50000p
[14] locate raminit :: 60000p
[15] init32p raminit :: GROUP FLAT_DATA
[16] fixup selector start_data = group FLAT_DATA
[17] fixup far32 start_code = _main
[18] fixup far32 start_stack = start_tos
[19] fixup far32 start_init = segment raminit
[20] gdt GDTSYS[3..64] :: reserve
[21] idt IDTSYS[0..64] :: reserve
[22] tss386 tss386_sys :: cs:eip=_boot ss:esp=start_tos
[23] gdt GDTSYS :: *
```

No unresolved symbols found in application:

```
0 unresolved symbols
```

Segment map:

```
Segment map

 Sel:Offset   Length   Address    Name      Class Group      Mem
0210:00040000 00000000 00040000P _DATA     DATA  FLAT_DATA  ROM
0210:00040000 00000000 00040000P _BSS      BSS   FLAT_DATA  ROM
0210:00040000 00000066 00040000P .DATA     DATA  FLAT_DATA  RAM
0210:00040070 00000004 00040070P .BSS      UDATA FLAT_DATA  RAM
0210:00040080 00000009 0040080P  .UDATA    DATA  FLAT_DATA  RAM
0218:00040090 00001000 0040090P  _STACK    STACK            ROM
0008:00000000 00000230 00041090P GDTSYS                     ROM
0010:00000000 00000208 000412c0P IDTSYS                     ROM
0220:00000000 00000068 000414c8P TSS386_SYS                 ROM
0208:00050000 000000d3 00050000P INIT_TEXT CODE  FLAT_CODE  ROM
0208:000500d4 00000000 000500d4P _TEXT     CODE  FLAT_CODE  ROM
0208:000500e0 00000312 000500e0P .TEXT     CODE  FLAT_CODE  ROM
0228:00000000 000000bc 00060000P RAMINIT                    ROM
```

Global descriptor table information (partial):

```
GDT[0..69] GDTSYS at 00041090P

Entry  Sel:Offset  Name       Class   Group
_____________________________________________________________

GDT[0] 0000  Empty           00000000L Lim=00000H DPL=0 gbp av
GDT[1] 0008  Data WR         00041090L Lim=0022fH DPL=0 gbP av
       :00000000 GDTSYS

GDT[2] 0010  Data WR         000412c0L Lim=00207H DPL=0 gbP av
       :00000000 IDTSYS

GDT[3] 0018  Reserved
       .
       .
       .
GDT[65] 0208  Code RD        00000000L Lim=fffffH DPL=0 GBP av
       :00050000 INIT_TEXT   CODE      FLAT_CODE
       :000500d4 _TEXT       CODE      FLAT_CODE
       :000500e0 .TEXT       CODE      FLAT_CODE

GDT[66] 0210  Data WR        00000000L Lim=fffffH DPL=0 GBP av
       :00040000 _DATA       DATA      FLAT_DATA
       :00040000 _BSS        BSS       FLAT_DATA
       :00040000 .DATA       DATA      FLAT_DATA
       :00040070 .BSS        UDATA     FLAT_DATA
       :00040080 .UDATA      DATA      FLAT_DATA

GDT[67] 0218  Data WR        00000000L Lim=fffffH DPL=0 GBP av
       :00040090 _STACK      STACK

GDT[68] 0220  Avail 386 TSS 000414c8L Lim=00067H DPL=0 gBP av
       :00000000 TSS386_SYS

GDT[69] 0228  Data WR        00060000L Lim=000bbH DPL=0 gbP av
       :00000000 RAMINIT
```

Interrupt descriptor table (partial):

```
IDT[0..64] IDTSYS at 000412c0P

IDT[0]      00  Reserved
IDT[1]      01  Reserved
IDT[2]      02  Reserved
IDT[3]      03  Reserved
    .
    .
    .
IDT[64]     40  Reserved
```

TSS information:

```
Initial TSS386:  TSS386_SYS

gdt[68] 0220 Avail 386 TSS  000414c8L Lim=00067H DPL=0 gBP av

EAX=00000000    EBX=00000000    ECX=00000000    EDX=00000000
ESI=00000000    EDI=00000000    EBP=00000000
DS=0000         ES=0000         FS=0000         GS=0000
LDTR=0000       LINK=0000
CS:EIP=0208:00050014
SS:ESP=0218:00041090
SS0:ESP0=0000:00000000
SS1:ESP1=0000:00000000
SS2:ESP2=0000:00000000
EFL=00000000 [ ac vm rf nt IOPL=0 of df if tf sf zf af pf cf ]
CR3=00000000 [ PDBR=0 pcd pwt ]
IO_MAP=0000
TRAP=0
```

Files created specified here.

Public symbol information summary:

```
Translating debug symbolics


Debug Symbolics Translation Complete:

Modules. . . . . 4
Procedures . . . . 8
Public Symbols . . 21
Source Lines . . . 184
Total Symbols. . . 51
Types. . . . . 552

Linking complete, No errors or warnings.
```

End of map file.

Toolchains and Memory Segmentation Models

1

Table 1-1: 32-bit Protected-Mode Applications

Compiler	Assembler	Memory Model(s) for Applications
Borland 32-bit C/C++	Borland TASM	Flat
MetaWare High C	Microsoft MASM	Flat or Segmented
Microsoft 32-bit C/C++	Microsoft MASM	Flat
Watcom 386 C/C++	Watcom WASM or Microsoft MASM	Flat or Segmented

Troubleshooting Location Problems

1. Assembly language segments located incorrectly

If a segment defined in assembly language gets located a few bytes beyond where you specify with the ***LOCATE*** command in a **.cmd** file, define the segment as paragraph-aligned in your assembly code to prevent such dislocation.

2. Truncated segments

If the linker undesirably truncates a segment or segments that have been padded with zero-bytes (or padded in some other way), use the ***INTEGRITY*** command (defined in Chapter 3) to direct the linker to preserve those padded bytes and not truncate those segments.

2. 32-bit Protected-Mode Applications

2

Chapter Contents

Overview

This chapter covers preparing and locating 32-bit protected-mode applications. There are four examples of such applications supplied with the linker and they can be found in the following subdirectories:

samp\bcc32p (Borland)
samp\hc32p (MetaWare)
samp\msc32p (Microsoft)
samp\wcc32p (Watcom)

This chapter will discuss the example in **samp\msc32p** to illustrate how you can prepare and link your own 32-bit protected-mode application.

32-bit Protected-Mode Example

Step 1—Compile Using Microsoft Tools

See also: For other tool chains, see the *Tools* section of this chapter.

We have used makefiles to create the sample programs included in all **samp** subdirectories. For the sake of illustration in this chapter, we will discuss the program found in **samp\msc32p**.

The makefile can produce two applications. The `ABSOLUTE` makefile flag determines which application is generated. One is an application which can be downloaded by a debugger for debugging. It assumes there is already a monitor on the target board and that the monitor is already in protected mode.

The second application creates a 32-bit hex file for writing into flash. It runs from the restart address on the Intel 386EX evaluation board.

The example command file described in this chapter is for a ROMmable application which will start executing at the restart address. This command file is shown below.

Example Command File csamp.cmd

```
// Command file for building sample program hex file

object start.obj unpack.obj
object cmain.obj cutils.obj io.obj
hex386
flat
cpu 386ex
debug
create SYS_GDT
create raminit
gdt SYS_GDT::group FLAT_CODE group FLAT_DATA _BOOT RAMINIT *
init32p raminit :: group FLAT_DATA
entry cs:eip = _startup
attribute class CODE :: use32 er
attribute segment _BOOT :: use16 er
attribute segment raminit :: USE32 er
attribute group FLAT_DATA :: USE32 rw
locate segment _BOOT :: 0F0000P
locate group FLAT_DATA :: 00010000P
locate segment .TEXT SYS_GDT :: 0080000P
locate segment raminit :: 0091000P
fixup physical link_gdt = segment SYS_GDT
```

Step 2—Create a Command File

Use an ASCII text editor to create a **.cmd** command file with the commands that direct the linker to locate your application. The command file illustrated above is generated by the makefile in **samp\msc32p.** We will begin to explain it in detail in numbered paragraphs just below. The full story of the linker's commands and how to construct command files with them is in Chapter 5.

1. Note that comments begin with double slashes and end with a carriage return.

2. The order in which commands occur in the command file is significant. You can think of the linker as though it is an interpreter that processes each command as it reads it in. To help you get started with ordering commands, we provide general guidelines for command ordering in **.cmd** files under the *Command File Organization* section of Chapter 5. In general, place I/O commands first (the first three commands in the previous example are I/O commands) and location and table-construction commands last.

3. ***OBJECT*** identifies the object files which make up your application. You may specify more than one object file per ***OBJECT*** command by separating object filenames with commas or spaces. The linker will process the object files in the order they are specified in the command file.

4. ***HEX386*** directs the linker to generate an Intel 32-bit hex file. By default the name of the hex file is the same as the **.cmd** file.

5. ***FLAT*** specifies that the input files have been compiled to generate flat code. Except for special segments such as `GDT`, `IDT`, etc., all segments are based at `0P` and their limit is the processor's highest address.

6. ***CPU*** specifies the target CPU to the linker. A list of possible values in the 80x86 family is given in Chapter 5 under ***CPU***. This value is used by the linker to produce optimal output.

7. ***DEBUG*** directs the linker to output symbolic debugging information that is essential for a debugger. The symbolics are placed in the actual absolute output file.

8. ***CREATE*** creates a segment. It is useful for creating segments for protected-mode structures (`TSS, GDT, IDT,` and `LDT`), RAM-initialization code (***RAMINIT***), and for alias segments that are used to accommodate preexisting segments.

9. ***GDT*** is used to fill in the Global Descriptor Table (`GDT`). The command assumes that a segment to hold the `GDT` (**sysgdt** in this case) has already been created. For this example, the segment was created by a use of the linker's ***CREATE*** command, though it is

equally possible to explicitly create the segment with a directive in an assembly file (see the ***Features*** section of Chapter 1).

In this example the command starts `GDT` slots at slot 3. Group `FLAT_CODE` is assigned `GDT[3]`, group `FLAT_DATA` is assigned `GDT[4]` and so on. Other segments not explicitly specified are assigned following `RAMINIT` and are indicated by the asterisk in the list.

Other sample applications provided will reserve slots in the `GDT` for an application such as a monitor which may be already on the target board.

10. The `INIT32P` *`seg_name`* `::` *`seg_list`* command specifies that the linker will pack data from the segments in *`seg_list`* into segment *`seg_name`*. This is useful for read-write data that you want to be initialized at boot-up time. If you don't have some place to store initialized data in ROM and then copy it into RAM, all RAM-based data in your program will be zeroed out. There are macros provided with the linker, in the file **link.inc**, that unpack and copy data from ROM to RAM. For more details on these macros, see the *Features* section of Chapter 1.
11. ***ENTRY*** specifies the entry point for the application.
12. ***ATTRIBUTE*** alters the attributes of a protected-mode segment.
13. ***LOCATE*** plays perhaps the most crucial role among all the linker commands, for obvious reasons. This command tells the linker explicitly to absolutely locate one or more segments, which in turn may cause other segments to be located. This kind of ripple location effect results from the linker's relative ordering of segments: Once the first segment in an ordered collection of segments is located, all subsequent segments as determined by the linking process fall into place.

    You may find it useful to first locate all segments at a starting memory location (e.g., `LOCATE * :: 8000P`). The map file produced by the linker can be used to see the names of the segments and the default ordering of segments.

In this example the startup code found in segment `_BOOT` is located at `F0000P`. Data segments are located at `10000P` while segments `TEXT` and `SYS_GDT` are located beginning at `80000P`. Finally, segment `RAMINIT` is located at `91000P`.

14. ***FIXUP*** provides a way for you to modify data found in segments. The example here installs an address of `SYS_GDT` in `link_gdt`.

Step 3—Invoke the Linker

To create a located application, use the following syntax at the DOS prompt or in a makefile:

```
LINK filename
```

where *`filename`* is the command file (with default extension **.cmd**) that contains the linker commands, as in the following example:

```
link csamp
```

If the linker links and locates your application as specified without error, the output file(s) that you request with the commands ***ABS86***, ***ABS386***, ***BINARY***, ***HEX86***, and ***HEX386*** are created, and a map file ***filename*.map** is also produced. If the linker encounters any errors while trying to locate your application, the only output file is the **.map** file, which shows all warning and error messages generated during linking. Parts of the **.map** file produced for our example 32-bit protected-mode application here are given below, with annotations. There is a more complete **.map** file in Chapter 1.

Example Map File

[In CSAMP.MAP] Segment map that shows located segment addresses

```
Segment map

 Sel:Offset         Length Address   Name Class   Group       Mem

0020:0001000000000066  00010000P .DATA  DATA   FLAT_DATA    RAM
0020:0001007000000004  00010070P .BSS   UDATA  FLAT_DATA    RAM
0020:0001008000000009  00010080P .UDATA DATA   FLAT_DATA    RAM
0018:00080000000003e1  00080000P .TEXT  CODE   FLAT_CODE    ROM
0008:0000000000000038  000803e1P              SYS_GDT      ROM
0030:00000000000000bc  00091000P  RAMINIT                   ROM
0028:0000000000000303  000f0000P _BOOT  BOOT                 ROM
```

[In CSAMP.MAP] Initial GDT

```
GDT[0..6] SYS_GDT at 000803e1P

Entry  Sel:Offset   Name    Class   Group

GDT[0] 0000  Empty  00000000L Lim=00000H DPL=0 gbp av
GDT[1] 0008  Data WR000803e1L Lim=00037H DPL=0 gbP av
       :00000000 SYS_GDT

GDT[2] 0010  Empty  00000000L Lim=00000H DPL=0 gbp av
GDT[3] 0018  Code RD00000000L Lim=fffffH DPL=0 GBP av
       :00080000 .TEXT     CODE      FLAT_CODE

GDT[4] 0020  Data WR00000000L Lim=fffffH DPL=0 GBP av
       :00010000 .DATA     DATA      FLAT_DATA
       :00010070 .BSS      UDATA     FLAT_DATA
       :00010080 .UDATA    DATA      FLAT_DATA
```

```
GDT[5] 0028  Code RD000f0000L Lim=00302H DPL=0 gbP av
       :00000000 _BOOT     BOOT

GDT[6] 0030  Code RD00091000L Lim=000bbH DPL=0 gBP av
       :00000000 RAMINIT
```

Borland Tools

Here are the controls to use when preparing your 32-bit protected-mode application with Borland tools and the linker.

Borland C/C++ Compiler

Use these controls with the Borland compiler:

`-v` Debug information.

`-3` Generate 32-bit 80386 protected-mode instructions.

`-O-` Disable optimization. You may remove this switch when the module has been debugged.

`-c` Don't link.

Example invocation:

```
bcc -v -3 -O- -c  cmain.c
```

Borland Turbo Assembler

Use these controls with the Borland assembler:

`/Zi` Provide debug information.

`/mx` or `ml` Treat symbols as case sensitive.

Example invocation:

```
tasm32 /Zi /mx b32fpbcc.asm
```

MetaWare Tools

Here are the controls to use when preparing your 32-bit protected-mode application with MetaWare tools and the linker.

MetaWare Compiler

Use these controls with the MetaWare compiler:

`-debug` Provide debug information

`-pro` Define a profile file

The file **cv.pro** is a MetaWare profile file that contains compiler controls that ensure that the proper symbolics for your debugger are produced. An example copy of this file is located in the **samp/hc32p** subdirectory where you installed the linker.

Example invocation:

```
hcdx86 cmain.c -mm large -debug -pro cv.pro
```

Use these controls for versions 3.0 or greater:

`-g` Provide debug information

`-Hpro=cv.pro` Define a profile file

Example invocation:

```
hc386 -g -c -Hpro=cv.pro cmain.c
```

Microsoft Tools

Here are the controls to use when preparing your 32-bit protected-mode application with Microsoft tools and the linker.

Microsoft C/C++ Compiler

Use these controls with the Microsoft compiler:

`/Zi` Include symbolic information. Versions 8 and the Visual C++ compiler versions use **`/Z7`** to perform this function.

`/Od` Disable optimization. You may remove this switch when the module has been debugged. It is even possible to leave this switch out, but we recommend you do this only after you are comfortable using your debugger.

`/c` Compile only—do not link.

`/Gs` Remove run-time stack probes.

Example invocation:

```
cl  /Zi /Od /c cmain.c
```

Microsoft ML

Use these controls with the Microsoft assembler:

/Zd Include line number information in object file.

/Zi Generate Codeview symbolics in object file.

/Cp Make all symbols case sensitive.

/c Compile only—do not link.

Example invocation:

```
ml /Zd /Zi /Cp /c b32fpmsc.asm
```

Watcom Tools

Here are the controls to use when preparing your 32-bit protected-mode application with Watcom tools and the linker.

Watcom C/C++

Use these controls with the Watcom compiler:

/s Remove stack overflow checking.

/d2 Create debug information.

/hc Create Codeview-compatible debug information.

Example invocation:

```
wcc386 /hc /s /d2 cmain.c
```

Watcom WASM

Use this control with the Watcom assembler:

-d1 Create debug information.

Example invocation:

```
wasm -d1 b32pwcc.asm
```

3. Command Reference

Chapter Contents

Overview

This chapter describes the linker's commands and how you can use them to link and locate your application in precisely the way that you want it located.

The chapter begins with a description of the **.cmd** command file that you build to contain commands that the linker follows to locate your application. Then there is a set of command-ordering guidelines that are intended to assist you in organizing the commands in your **.cmd** command file. They are more heuristic in nature than they are hard-and-fast rules.

Then you will find a summary of the syntax elements in commands, followed by a summary list of the linker's commands. The remainder of the chapter consists of an alphabetically ordered command-reference section containing a detailed explanation of each command.

The .CMD Command File

The linker uses a sequential command file to control processing action. Here are some of its general characteristics:

- The default command-file extension is **cmd**.
- The ***VERBOSE*** command, which is used to provide extra information to you about what the linker is doing, can occur anywhere in the command file. Turn on verbose mode by adding the keyword ***ON*** to the command, and off by adding ***OFF***. Below is a part of a .**map** file that exemplifies the sort of messages that you receive in verbose mode:

```
[13] verbose on
[14] create SYS_GDT
    >>>  Created segment: SYS_GDT.
[15] create raminit
    >>>  Created segment: RAMINIT.
[16] gdt SYS_GDT[1..2] :: reserve
    >>>  Defining SYS_GDT as a gdt
[17] gdt SYS_GDT::_TEXT group DGROUP _BOOT RAMINIT *
    >>>  FLAT_CODE assigned GDT[3]
    >>>  DGROUP assigned GDT[4]
    >>>  _BOOT assigned GDT[5]
    >>>  RAMINIT assigned GDT[6]
    >>>  FLAT assigned GDT[7]
    >>>  SYS_GDT assigned GDT[1]
[18] init32p raminit :: _data _bss
    >>>  Defining RAMINIT as a 32-bit protected-mode
            RAM init table
    >>>    _DATA placed into table.
    >>>    _BSS placed into table.
```

- Commands that locate classes locate the entire class contiguously according to the linker's default ordering of segments within the class. If you want to locate a segment separately from the rest of its class, you must place locating commands for that segment

before commands that locate the rest of the class or use the EXCEPT keyword.

- With the exception of public symbol names, whose characters must exactly match in case the names used to declare them in application files, all names, identifiers, prefixes, and suffixes in command files are not case sensitive. Thus, the following examples are equivalent:

```
CREATE MY_DATA :: LIMIT=0X50
create my_data :: limit=0x50
```

  This may cause problems if you have symbols that differ only in case and you compile with a case-sensitivity switch on.

- Blank lines and other white spaces are ignored and can be used however you want.
- Maximum command-line length is 222 characters.
- Commands may span multiple lines. To continue a command on subsequent lines, use a plus sign (+) as the first character on each continuation line:

```
tss386 tss_xxx :: cs:eip=main, ds=data_seg,
+                 fl.if=0x1
```

- Comments can be placed anywhere in the command file. Use double slashes to start a comment; a comment ends at the end of the line that it starts on:

```
// This is a sample comment line
cpu 386     //This is another sample comment,
            //which spans two lines
```

- Command files should be structured according to the command-grouping guidelines given in the next section of this chapter. These guidelines are not hard and fast rules for command-file construction, as some of the example command files in earlier chapters testify to. In learning to use the linker's 44 commands, you can use the guidelines to provide order to an otherwise seemingly random command-file construction process.

Command File Organization

Recommended Ordering of Commands

We recommend that you order the commands in the **.cmd** command file according to the following groupings. Place commands in Group-1 first, then place commands in Group-*N*+1 after commands in Group-*N*. Commands within the same group can be ordered in any way you want.

Though these are just recommended guidelines, we strongly urge that you learn to build your own linker command files by following them. Don't be surprised if you notice that the example command files discussed in Chapters 2–4 don't follow these guidelines strictly. They abide by the essential rules, but may diverge for less important ones.

Only commands in Group-1 absolutely must be placed before commands in Group-3 and above, and Group-6 commands should occur last. The Group-0 command (***VERBOSE***) can occur freely throughout the command file.

See also: For more on the ***VERBOSE*** command, see the ***VERBOSE*** section of this chapter.

Grouping Guidelines

Follow these grouping guidelines when you build your command file:

Group-0 (Linker Debug Information)
VERBOSE

Group-1 (Input)
BC5LIB
CPU
LIBRARY
OBJECT

Group-2 (Output)

ABS86
ABS286
ABS386
BINARY
DEBUG
ENTRY
HEX86
HEX386
PRINT

Group-3 (Segment Creation, Definition, and Alteration)

ALIAS
ATTRIBUTE
CREATE
FLAT
FIXUP
INIT16P
INIT16R
INIT32P
PAGEDIRECTORY
PMODE
RAM
RMODE

Group-4 (Segment Location)

BASE
INTEGRITY
LOCATE
ROMBASE
ROMMOVE

Group-5 (Protected-Mode Structures)

CALL286
CALL386
INT286
INT386
TASKGATE
TRAP286
TRAP386
TSS286
TSS386

Group-6 (Table Constructors)

GDT
IDT
LDT
PAGETABLE

Command Syntax

Table 3-1: Command Syntax Elements

Element	Meaning
*	Signifies all other segments that have not already been explicitly located, modified, etc.
|	Separates mutually exclusive alternatives.

( )	Enclose alternative entries (separated by "|"), as in the following example:
	`CS=(number | seg_name | pub_sym)`
	is equivalent to,
	`CS=number | CS=seg_name | CS=pub_sym`

[ ]	Enclose optional entries.
addressL	hex_numL {linear address}
addressP	hex_numP {physical address}
asn_expr	GROUP gname [:offset] [(+|-) adjust] SEGMENT sname [:offset] [(+|-) adjust] GROUPOF pubsym [:offset] [(+|-) adjust] SEGMENTOF pubsym [:offset] [(+|-) adjust] pubsym [(+|-) adjust] constant [:offset] [(+|-) adjust]
assign	field=ptr_value
assign_list	See Tables 5.9 and 5.10 under ***TSS286*/*TSS386***.

Table continued on next page.

Table 3-1: Command Syntax Elements (continued)

Element	Meaning
attribute_list	See Table 5.3 under ***ATTRIBUTE*** in this chapter.
class_name	Character string that identifies a class.
cpu_name	See Table 5.5 under ***CPU*** in this chapter.
dec_num	Decimal number.
dir_name	Name of segment where page-table directory is located.
entry_list	See Table 5.6 under ***ENTRY*** in this chapter.
file_list	filename [[,] filename]*
filename	DOS filename with optional extension.
gate_options	DPL=number | COUNT=number | (PRESENT|NOTPRESENT) | ENTRY=ptr_value
group_name	Character string that identifies a group.
hex_num	Hex number. Must have prefix 0x (or 0X) or suffix H (or h). Numbers that begin with a letter (a..f) must have a zero (0) prefix. If more than eight numbers are given, the eight least significant digits are used.
kind	See Table 5.7 under ***FIXUP*** in this chapter.
num_value	(OFFSETOF pub_sym) | number
range	[number1[..number2]] Beginning and ending brackets are required. If number2 is omitted, range has length 1 starting at number1.

Table continued on next page.

Table 3-1: Command Syntax Elements (continued)

Element	Meaning

seg_list	Segment list containing segments, classes, and groups, arranged in any order and used as many times as you want. Specify elements in any of the following ways (optional commas can be used to separate entries, as in first line below) :
	`seg_name_opt [[,] seg_name_opt]*`
	`GROUP group_name [EXCEPT seg_name_opt [seg_name_opt]*]`
	`CLASS class_name [EXCEPT seg_name_opt [seg_name_opt]*]`
	`* [EXCEPT seg_name_opt [seg_name_opt]*]`

seg_name	Character string that identifies a segment.
seg_name_opt	[SEGMENT] seg_name

seg_value	`SEGMENTOF pub_sym`
	`GROUP group_name`
	`seg_name_opt`

selector:offset	A logical address consisting of two hex numbers separated by a colon. Hex-number suffix or prefix is not required, that is, any number before or after a colon is automatically interpreted as a hex number.

Table 3-2: Keywords Used in Linker Commands

Keyword	Meaning
CLASS	Indicates following name is a class name
COUNT	Gate-descriptor word count
DPL	Gate-descriptor privilege level
ENTRY	Gate entry point
EXCEPT	Indicates exclusion of following segment(s), class(es), or group(s)
GROUP	Indicates following name is a group name
LENGTH	Indicates following number is the number of bytes after the public symbol that PMODE or RMODE applies to
NOTPRESENT	Signifies gate-descriptor present flag is false
OFF	Signifies end of verbose mode
OFFSETOF	Indicates offset of following public symbol
ON	Signifies start of verbose mode
PRESENT	Signifies gate-descriptor present flag is true
RESERVE	Indicates descriptor-table entries are reserved
SEGMENT	Indicates following name is a segment name
SEGMENTOF	Indicates segment of following public symbol

Command Syntax Summary

```
ABS86  [filename]
ABS286  [filename]
ABS386  [filename]

ALIAS seg_name_opt1 :: (seg_name_opt2   |
                        (GROUP group_name ))

ATTRIBUTE seg_list :: attribute_list

BASE seg_list :: addressL

BC5LIB

BIN[ARY]  [filename]

CALL286 seg_name :: gate_options

CALL386 seg_name :: gate_options

CPU cpu_name

CREATE seg_name [:: attribute_list]

DEBUG

ENTRY entry_list

FIXUP kind  pubsym [+ offset] = asn_expr

FLAT

GDT seg_name [range]  [:: (RESERVE | seg_list )]

HEX [filename]

IDT seg_name [range]  [:: (RESERVE | seg_list )]

INIT16P seg_name :: seg_list

INIT16R seg_name :: seg_list

INIT32P seg_name :: seg_list

INT286 seg_list :: gate_options
```

```
INT386 seg_list :: gate_options

INTEGRITYseg_list

LDT seg_name [range ]  [:: (RESERVE | seg_list )]

LIBRARY * l file_list

LOCATE seg_list :: (addressL  | addressP)

OBJECT file_list

PAGEDIRECTORY dir_name [range] :: seg_list

PAGETABLE dir_name :: seg_list

PMODE seg_list   |
      pub_sym1 [[to pub_sym2 ] | [LENGTH number ]]

PRINT

RAM seg_list

RMODE seg_list   |

      pub_sym1 [[to pub_sym2 ] | [LENGTH number ]]

ROMBASE seg_list :: addressP

ROMMOVE seg_list :: addressP

TASKGATE seg_list :: gate_options

TRAP286 seg_list :: gate_options

TRAP386 seg_list :: gate_options

TSS286 seg_list :: assign_list

TSS386 seg_list :: assign_list

VERBOSE ON | OFF
```

Italicized Syntax Elements

For an explanation of the italicized syntax elements (e.g., *seg_list)*, refer to Table 3-1 *Command Syntax Elements*.

Command Reference

These commands are described in this section:

ABS386, ABS286, and ABS86 Create an .**abs** output file and/or change the file's name.

ALIAS Creates protected-mode aliases.

ATTRIBUTE Changes the attributes of a protected-mode segment.

BASE Changes the descriptor base for a segment or group to a linear address.

BC5LIB Informs the linker that Borland v5.0 libraries are being linked.

BIN Creates a binary output file and/or changes the name of the file.

CALL286, CALL386 Set segments as call-gate descriptors.

CPU Specifies the processor.

CREATE Creates a segment.

DEBUG Generates symbolic information.

ENTRY Sets initial register values.

FIXUP Modifies your application.

FLAT Makes segments have a base of zero and a limit of the maximum processor address.

GDT Builds the protected-mode GDT table.

HEX86 and HEX386 Create a **hex** output file and/or rename the file.

IDT Builds the protected-mode IDT table.

INIT16P, INIT16R, INIT32P Pack data from RAM segments into ROM segments.

INT286, INT386 Set up interrupt-gate descriptors.

INTEGRITY Forces the linker to locate and include any empty spaces or padding within a segment.

LDT Builds the protected-mode LDT table.

LIBRARY Specifies library files to be linked.

LOCATE Locates segments, classes, or groups in ROM or RAM.

OBJECT Specifies object files to be linked.

PAGEDIRECTORY Defines segments as page directories and page tables.

PAGETABLE Specifies segments to be mapped through the page directory.

PMODE Builds mixed-mode applications.

PRINT Prints public symbol information in the map file.

RAM Specifies segments to not be placed in the output file.

RMODE Builds mixed-mode applications.

ROMBASE Sets the base address of ROM.

ROMMOVE Increases hex-record addresses.

TASKGATE Sets up task-gate descriptors.

TRAP286 , TRAP386 Set up trap-gate descriptors.

TSS286, TSS386 Set segments and specify TSS fields.

VERBOSE Prints additional information to the map file.

ABS386, ABS286, and ABS86

Create an **.abs** output file and/or change the file's name.

```
ABS386  [filename]
ABS286  [filename]
ABS86   [filename]
```

This command creates an **.abs** output file and can also be used to change the file's name, which by default is the same as the **.cmd** command file input to the linker.

- By default, no absolute, binary, or hex file is output. You must specify an output command to generate output.
- Output commands can be used one at a time, all together, or in any combination. Each command will generate one output file.

Examples

```
abs86 csamp.abs

abs386

abs286 my_file.out
```

ALIAS

Creates protected-mode aliases.

```
ALIAS  [SEGMENT] seg_name1  ::
     (([SEGMENT] seg_name2)   |
     (GROUP group_name))
```

This command makes *seg_name1* a protected-mode alias of *seg_name2* or *group_name*.

- *seg_name1*'s base and limit are set to the base and limit of *seg_name2*.
- *seg_name1* can have its own attributes and selector.
- Any data previously located in *seg_name1* is lost.
- This command can be used to write into a code segment.
- The attributes of an alias segment are `RW` (read/write).

Example

```
create ldt_alias :: limit=0ffffh

alias ldt_alias :: sys_ldt0

ldt sys_ldt0[1] :: ldt_alias
```

ATTRIBUTE

Changes the attributes of a protected-mode segment.

ATTRIBUTE *seg_list* :: *attribute_list*

Use this command is used to alter the attributes of a protected-mode segment.

- Only the attributes in *attribute_list* are changed. All other descriptor fields are left intact.
- Use the items in *Segment Attributes* on the next page to create an *attribute_list*. Items may be used repeatedly and in any order. Separate entries with commas or spaces.

Segment-type Abbreviations

RO	Read only, data segment
RW	Read/Write, data segment
ROED	Read only/Expand down, data segment
RWED	Read/Write/Expand down, data segment
EO	Execute only, code segment
ER	Execute/Read, code segment
CEO	Execute only/Conforming, code segment
CER	Execute/Read/Conforming, code segment

NOTE: You cannot use the LIMIT attribute to decrease the size of a segment that is created by your application.

Segment Attributes

Attribute	Descriptor Correspondence
`DPL=`*`number`*	Set the privilege level in descriptor for segment
`LIMIT=`*`number`*	Set segment limit
`LIMIT+=`*`number`*	Increase current limit
`BYTEGRAIN | PAGEGRAIN`	Byte or page granularity used for limit in descriptor
`PRESENT | NOTPRESENT`	Present bit in descriptor
`AVAILABLE | NOTAVAILABLE`	Available bit in descriptor
`USE32 | USE16`	16- or 32-bit segment
`RO | RW | ROED | RWED | EO | ER | CEO | CER`	Set segment type in descriptor
`PAGE.PRESENT | PAGE.NOTPRESENT`	Page-present bit
`PAGE.RO | PAGE.RW`	Read only or Read Write page
`PAGE.USER | PAGE.SUPER`	User or supervisor protection level
`PAGE.ACCESSED | PAGE.NOTACCESSED`	Page accessed bit
`PAGE.DIRTY | PAGE.NOTDIRTY`	Page dirty bit

Examples

```
attribute init_text ::    limit=1000H,
+ dpl=0,
+ present
attribute group cgroup :: use32
```

BASE

Changes the descriptor base for a segment or group to a linear address.

BASE *seg_list* :: *addressL*

This command forces the descriptor base for a segment or group to be the linear address given.

- The segment's physical and linear addresses are not affected, but the offset of its logical address is shifted.
- This is typically used with flat-model applications to make the offset into a segment match its physical address.

Example

```
base init_text :: 4000L
```

BC5LIB

Informs the linker that Borland v5.0 libraries are being linked.

`BC5LIB`

This command is used to inform the linker that Borland v5.0 libraries are being linked by the linker. Borland v5.0 libraries contain extra fields which must be processed. Unfortunately, there are no fields prior to the extra fields to distinguish the Borland library from other compilers' libraries.

Specify this command prior to any ***LIBRARY*** commands containing Borland v5.0 libraries. If you are linking libraries from other compilers, specify those libraries prior to the BC5LIB command.

Example

```
BC5LIB
```

BINARY

Creates a binary output file and/or changes the name of the file.

```
BINARY [filename]
```

This command creates a **.bin** binary output file and can also be used to change the name of the file, which by default is the same name as the **.cmd** file that is the linker's input command file.

The binary (**.bin**) file produced by this command is a raw dump of the application from the lowest address in the application to the highest address in the application. Any gaps in this memory range not used by the application are filled with `0xFF`. There are no header records in the file, just the data.

- By default, no absolute, binary, or hex file is output. You must specify an output command to generate output.
- Output commands can be used one at a time, all together, or in any combination. Each command will generate one output file.

Examples

```
binary test_app.bin

binary

binary csamp.bnr
```

CALL286 and CALL386

Set segments as call-gate descriptors.

```
CALL286 seg_name  :: gate_options
CALL386 seg_name  :: gate_options
```

These commands set segments *seg_name* as a call-gate descriptor.

- Use the items in Table 3-3 below to form your *gate_options*. Items may be used in any order and may be repeated, separated by a space or comma.
- DPL and COUNT both default to zero.
- PRESENT | NOTPRESENT defaults to PRESENT.
- ENTRY defaults to the address stored at offset 0 within segment *seg_name*. You can use support macros to predefine these values in your assembly module. See the macro file **link.inc,** which is located in the directory where you installed the linker (see *Features* in Chapter 1 for a list of the macros).
- If the macros are not used, the segment created must be at least 16 bytes long.
- COUNT is the number of DWords which will be copied from the caller's stack to the stack of the called procedure.

Table 3-3: Gate Attributes for CALL286 and CALL386

Gate Attributes	Descriptor Correspondence
DPL=number	DPL bits
COUNT=number	Word count
PRESENT | NOTPRESENT	Present bit
ENTRY=ptr_value	Code location the gate vectors to

Example

Place the following gate definition in your assembly file. The 0 parameter indicates the procedure has no parameters.

```
def_gate     system, 2, 0      ;in assembly invoke macro
```

Place the following C code in one of your source files:

```
void far system_gate (void)
procx ( )
}
   .
   .
   .
system_gate ()// Only the selector is used.
   // Descriptor contains
   // address for system_entry.
   .
   .
   .
}
system_entry ( )
{
}
```

The command syntax for the command file would be:

```
call386 system   // no options since options
                 // defined in macro
```

NOTE: The C code above assumes you are using a segmented memory model. In flat memory model, the C compiler will only generate near calls and the current selector value is assumed. To work around this limitation, "call" the `system_gate` from assembly language where you have more control of the object code generated.

CPU

Specifies the processor.

`CPU` *`cpu_name`*

This command is used to specify the exact processor of the target.

- This command must be placed near the beginning of the command file, before any segment location or manipulation commands.
- If this command is omitted, the linker defaults to a 386DX.
- Use the terms under *CPU Names* below to specify *`cpu_name`*.

CPU Names

Pentium				
486	486SX	486DX		
386	386SX	386DX	386EX	
376				
286				
188	C188	188EA	188EB	188EC 188XL
186	C186	186EA	186EB	186EC 186XL
88				
86				
V20	V30	V40	V50	

Examples

```
cpu pentium
cpu C186
```

CREATE

Creates a segment.

CREATE *seg_name* [:: *attribute_list*]

This command creates a segment with the given name and optional attributes.

- *Seg_name* must not conflict with any name of a segment already defined by the application.
- *Attribute_list* may contain any of the attributes in *Segment Attributes* under ***ATTRIBUTE*** in this chapter.
- Segments are placed in memory in the order in which they are created unless they are explicitly located otherwise by the user.

Example

```
create monitor_rom :: limit=2000H
```

DEBUG

Generates symbolic information.

DEBUG

DEBUG controls the generation of symbolic information. It has the following characteristics:

- If you want symbolics, you must use this command. By default, the linker does not generate symbolics.
- Symbolics are placed in the absolute output file produced by the linker. If no absolute file is requested, no debug information is processed.

Examples

debug

ENTRY

Sets initial register values.

```
ENTRY entry_list
```

This command sets initial register values. This command is useful if your application does not contain a TSS structure where you have specified the application start address and stack. The ***HEX86*** and ***HEX386*** commands need a starting address.

The fields listed below may be used with the ***ENTRY*** command.

ENTRY fields

```
CS:IP=(number:number | seg_name | pub_sym)
   CS=(number | seg_name | pub_sym)
   IP=number

CS:EIP=(number:number | seg_name | pub_sym)
   CS=(number | seg_name | pub_sym)
   EIP=number

SS:SP=(number:number | seg_name | pub_sym)
   SS=(number | seg_name | pub_sym)
   SP=number

SS:ESP=(number:number | seg_name | pub_sym)
   SS=(number | seg_name | pub_sym)
   ESP=number
```

Example

```
ENTRY cs:eip=main
```

FIXUP

Modifies your application.

```
FIXUP kind  pubsym [+ offset] = asn.expr
```

This command allows you to make simple modifications to your application while using the linker.

- When using the startup code supplied for use with your debugger, the label `cs_dgroup` must be zeroed for your application to build. Use the following to change the value of `cs_dgroup`:

```
fixup word cs_dgroup = group dgroup
```

- If you are not using your debugger's startup code, use ***FIXUP*** to change the values of the symbols to set up your stack.
- Use the information below for values for *`kind`*..

Fixup Kinds and their Byte Sizes

Kind	#Bytes	Description
BYTE	1	8-bit integer
DWORD	4	32-bit integer
FAR16	4	16-bit offset, 16-bit selector; pointer
FAR32	6	32-bit offset, 16-bit selector; pointer
LIMIT16	2	16-bit segment limit
LIMIT32	4	32-bit segment limit
LINEAR	4	32-bit linear address
NEAR16	2	16-bit offset only; pointer
NEAR32	4	32-bit offset only; pointer
PHYSICAL	4	32-bit physical address
SELECTOR	2	16-bit selector
TABLE	6	16-bit limit, 32-bit base
WORD	2	16-bit integer

Example

The following example assigns the physical address of `SYS_GDT` to a public symbol, `link_gdt`. This allows the application to copy the contents of the `GDT` created by the linker from ROM to RAM.

```
fixup physical link_gdt = segment SYS_GDT
```

FLAT

Makes segments have a base of zero and a limit of the maximum processor address.

FLAT

This command is used to make all segments in the application which are code or data have a base of zero and a limit of the maximum processor address. Segments which have special uses such as the GDT table or TSS structures are not affected by this command.

This command is useful if you have compiled your application using the flat memory model.

If you are only concerned with the start of the segment in relationship to the descriptor, use the ***BASE*** command.

Example

```
FLAT
```

GDT

Builds the protected-mode `GDT` table.

`GDT` *`seg_name`* [*`range`*] [:: (RESERVE | *`seg_list`*)]

This command is used to build the protected-mode GDT table.

- *`seg_name`* is where the table will be placed, and must be defined in your application (one way to do this is to use the macro ***def_tbl***, mentioned in Chapter 1 under *Features*, in your startup code) or created with the linker command ***CREATE***.

  If the only parameter used is *`seg_name`*, an empty table is created except for the default null and alias slots.

- *`range`* specifies the starting and optional ending index. The example below uses *`range`* to reserve slots for a monitor:

```
gdt sys_gdt[3..64] :: reserve
```

- `RESERVE` reserves the specified slots for system, monitor, or other uses. These slots are set to zero.
- When a range is not specified for the ***GDT*** command, the default starting slot is 3. `GDT[0]` is null, `GDT[1]` is the `GDT` alias, and `GDT[2]` is the `IDT` alias.
- Not all gates can be placed in all tables. Only the following can be placed in the `GDT` table:

  286/386 call gates
  Task gates

Example

The following example places the first segment found in the application at slot 5 of `tmp_gdt`. All other segments are placed in default order starting at slot 6.

```
gdt tmp_gdt[5] :: *
```

HEX86 and HEX386

Create a hex output file and/or rename the file.

`HEX86 [`*`filename`*`]`

`HEX386 [`*`filename`*`]`

This command creates a .**hex** output file and can also be used to change the name of the file, which by default is the same name as the .**cmd** file that is the linker's input command file.

- By default, no absolute, binary, or hex file is output. You must specify an output command to generate output.
- Output commands can be used one at a time, all together, or in any combination. Each command will generate one output file.

Examples

hex `prom.hex`

hex `c:\newapp\eprom.hex`

IDT

Builds the protected-mode `IDT` table.

`IDT` *`seg_name`* [*`range`*] [`::` (`RESERVE` | *`seg_list`*)]

This command is used to build the protected-mode `IDT` table.

- *`seg_name`* is where the table will be placed, and must be defined in your application (one way to do this is to use the macro **def_tbl**, mentioned in Chapter 1 under *Features*, in your startup code) or created with the linker ***CREATE*** command.
- *`range`* specifies the starting and optional ending index. The example below uses *`range`* to reserve slots for a monitor:

```
idt sys_idt[0..40] :: reserve
```

- `RESERVE` reserves the specified slots for system, monitor, or other uses. These slots are set to zero.
- `GDT[2]` is the `IDT` alias.
- Not all gates can be placed in all tables. The following are the gates that can be put in the `IDT` table:

  286/386 trap gates
  286/386 interrupt gates
  Task gates
- When a range is not specified for the ***IDT*** command, the default starting slot is `IDT[0]`.

Example

```
create int_114
int386 int_114 :: entry=timer_interrupt, DPL=0
IDT sys_idt[41] :: int_114
```

INIT16R, INIT16P, and INIT32R

Pack data from RAM segments into ROM segments.

```
INIT16R  seg_name  :: seg_list

INIT16P seg_name   :: seg_list

INIT32P seg_name   :: seg_list
```

These commands pack data from the segments in *seg_list*, which are to be located in RAM, and store the packed data in the ROM segment *seg_name*. Use these commands when you have constants or data that you want located in RAM and that need to be initialized at boot-up time.

- ***INIT16R*** applies to 16-bit real-mode applications.
- ***INIT16P*** applies to 16-bit protected-mode applications.
- ***INIT32P*** applies to 32-bit protected-mode applications.
- The data is stored in a packed form in ROM.
- The macros ***raminit_16r***, ***raminit_16p***, and ***raminit_32p***, which can be found in the file **link.inc**, unpack the data and copy it into the RAM segments in *seg_list*.

Example

The following example packs the data in all of the segments in class `data` and the segment `const` and stores the packed data in the segment `ram_init`. You can use the macros that are mentioned just above (in the last bulleted item) in your startup code to unpack and copy the data back to class `data` and segment `const` .

```
init16r ram_init :: class data segment const
```

Format of Data in ROM Segment

The list below shows the record types used to store RAM data in the ROM initialization segment:

Type	Description
00	End of table
11	Confirm real 16-bit table
12	Confirm protected 16-bit table
14	Confirm protected 32-bit table
20 *offset16/32 segment*	Load data pointer
30 *offset16/32*	Load data offset
40	Reserved
50-6F	Increment offset 1 to 32 bytes
70 *count16/32*	Zero fill
80-ff	Enumerated data block (1 to 128 bytes)

The linker will generate records in the order listed below for each segment in *seg_list*.

11 | 12 | 14 Initial record indicating the type of table.
20 Starting location in RAM of packed data.

70 Initialize RAM for *count* bytes to zero. The *count* is the length of the segment.

For the data stored in the segment:

30 | 50 Adjust offset to start of data block in RAM

80-ff Block of data—blocks of more than 4 bytes of zeros are not stored in ROM segment. Records 30 and 50 are used to adjust the offset to the next place in RAM where a data block is to be written. The record type (80-ff) indicates length of block written.

NOTE: Additional offset adjustment records and enumerated data records are written to account for the “fixup” of pointers in data to located segment and offset values.

After all segments in *seg_list*. have been processed, an end of table record (00) is written to the ROM segment.

The macros ***raminit_16r***, ***raminit_16p***, and ***raminit_32p*** read the above records doing the above actions described to read the data stored in ROM and write it into RAM.

The macro ***raminit_32p*** should only be called from 32-bit *USE32* code.

INT286 and INT386

Set up interrupt-gate descriptors.

```
INT286 seg_list :: gate_options
INT386 seg_list :: gate_options
```

These commands set up interrupt-gate descriptors and operate like the ***CALL*** commands shown in this chapter, with the following exceptions:

- The option `COUNT` is not used.
- When you define a interrupt gate using ***INT286*** or ***INT386***, you must include a command to place the gate in the IDT. If the ***def-gate*** macro is not used, the segment created must be at least 16 bytes long.

Example

The following example creates a segment, defines it as a interrupt gate, then places it in slot 50 of the IDT (this assumes that the segment `idtsys` has already been created):

```
create int_gt
int286 int_gt :: dpl=1 present
+       entry=init_text
idt idtsys[50]  :: int_gt
```

INTEGRITY

Forces the linker to locate and include any empty spaces or padding within a segment.

`INTEGRITY` *`seg_list`*

This command forces the linker to locate and include as part of your application any empty spaces or padding within a segment. This is quite helpful if your compiler writes extraneous data into segments that the linker isn't otherwise aware of.

- The effect of this command is to preserve any existing "padding" in segments.
- If you don't use this command, there are cases in which the linker suppresses a certain amount of padding when locating a segment.
- Empty space can occur, for example, in the segments that you declare for the `GDT` and `IDT`; use of this command would preserve all of that space.

Example

```
integrity *
```

LDT

Builds the protected-mode `LDT` table.

`LDT` *`seg_name`* [*`range`*] [:: (RESERVE | *`seg_list`*)]

This command is used to build the protected-mode `LDT` table.

- *`seg_name`* is where the table will be placed, and must be defined in your application (one way to do this is to use the macro ***def_tbl***, mentioned in Chapter 1 under *Features*, in your startup code) or created with the linker command ***CREATE***.
- *`range`* specifies the starting and optional ending index. The example below uses *`range`* to reserve the first ten slots:

```
ldt sys_ldt[0..9] :: reserve
```

- `RESERVE` reserves the specified slots for system, monitor, or other uses. These slots are set to zero.
- When a range is not specified for the ***LDT*** command, the default starting slot is `LDT[1]`.
- Not all gates can be placed in all tables. The following are the gates that can be put in the `LDT` table:

  task gates
  286/386 call gates

Example

The following example places `class code` in `ldt_1` starting at slot 2, then places all segments except those in `class code` into `ldt_2`, starting at slot 1:

```
ldt ldt_1[2] :: class code
ldt ldt_2 :: * except class code
```

LIBRARY

Specifies library files to be linked.

LIBRARY * | *file_list*

This command is used to specify library files to be linked by the linker. One or many files may be specified with this command. Separate library filenames with a space or comma. The linker will link libraries produced by the librarians provided with the Borland, MetaWare, Microsoft, and Watcom compilers.

The linker will only include modules from specified libraries to resolve symbols not resolved in previously listed object files. Place the ***LIBRARY*** command following all object files which might contain symbols which would be resolved by the libraries specified in the ***LIBRARY*** command.

Use the asterisk to specify that default libraries should be searched. By default, the default libraries are not searched. Default libraries are specified in object files.

Examples

```
LIBRARY *
LIBRARY mt.lib
```

LOCATE

Locates segments, classes, or groups in ROM or RAM.

LOCATE *seg_list* :: (*addressL* | *addressP*)

This command locates segments, classes, or groups in ROM or RAM, beginning at the given address.

- ***LOCATE*** assigns an address to the first segment in *seg_list*. If a class is given, the address is assigned to the first segment in the class.
- Once a segment is located, its location is permanent.
- Multiple instances of this command can be used in a command file, locating different segments, groups, or classes.
- Individual segments in groups cannot be located without the rest of the group.
- Individual segments in classes can be located by themselves.
- *addressL* is a linear address and must have an "*L*" suffix; linear ***LOCATE*** locates groups at specific addresses and maintains segments at adjacent linear addresses.
- *addressP* is a physical address and must have a "*P*" suffix; physical ***LOCATE*** places segments and allows nonadjacent addresses for same-group segments.

Examples

The following example first locates segment seg1, which let us assume is in class a_class, at 50000P, then locates the remaining segments in the a_class, and finally locates the segments in d_class:

```
locate seg1 class a_class class d_class :: 50000P
```

The next example uses the EXCEPT keyword to prevent seg1 from being located with the rest of its class. A separate ***LOCATE***

command or some other linker command would be needed to locate `seg1`.

```
locate class a_class except seg1 :: 50000P
```

NOTE: If a segment defined in assembly language gets located a few bytes beyond where you specify with the ***LOCATE*** command in a **.cmd** file, define the segment as paragraph-aligned in your assembly code to prevent such dislocation.

OBJECT

Specifies object files to be linked.

`OBJECT` *`file_list`*

This command is used to specify object files to be linked by the linker. One or many files may be specified with this command. Separate object filenames with a space or comma. The default extension is **.obj**.

The order of object files and the segments contained in those object files provides a default order for segments. The order and location of segments can be changed using the ***LOCATE*** command.

Examples

```
OBJECT test.obj
OBJECT usstart.obj, unpkrom.obj
```

PAGEDIRECTORY

Defines segments as page directories and page tables.

```
PAGEDIRECTORY dir_name [range ] :: seg_list
```

This command defines the given segment *dir_name* as a page directory and the segments in *seg_list* as page tables.

- You must create the segment *dir_name* with the ***CREATE*** command or in your startup code (see Chapter 1 under ***Features*** for the mention of a macro that can be used to create this segment) before you use this command.
- It allocates the exact position of each page table within the page directory.
- The full range of linear addresses used by the application must be accounted for.

For more information, see ***PAGETABLE*** in this chapter.

Example

```
pagedirectory dir_name[0] :: page_table
```

PAGETABLE

Specifies segments to be mapped through the page directory.

PAGETABLE *dir_name* :: *seg_list*

This command specifies that the segments in *seg_list* are to be mapped through the page directory *dir_name*.

The ***PAGEDIRECTORY*** *range* parameter defines the pagetable range, as in `table[1]`, `table[2]`, `table[3]`..., and is useful when you want to split your application into separate pieces, or if your application is large.

This is because the page table and the linear address of a segment are directly related—given a certain linear address, the physical address associated with a segment will be placed in a specific page table. The CPU controls this, and the linker cannot alter it.

However, by controlling where a segment is located you can control to some extent which page table it is associated with. This is important because the linker commands ***PAGEDIRECTORY*** and ***PAGETABLE*** set up the page tables, and if they do not set up a table that one of your segments is associated with, a fatal error will occur.

For those segments you do not locate explicitly, their location depends on the order in which they were created. So, if one of your segments ends up in the wrong table, you can put it in another table without explicitly locating it by creating it earlier in your application.

Examples

The first example below takes advantage of this feature. The range specified in the ***PAGEDIRECTORY*** command is [0], and as long as the page tables needed are consecutive, the linker sets them up. However, if you locate your segments so that you have segments associated with `table[0]`, then skip `table[1]` and have segments associated with `table[2]`, `table[2]` will not be set up.

In the second example below, if some of the segments are located at linear addresses `0f0000000L` through `0f00000003L`, then we need page `table[960]`.

```
pagedirectory dir_name[0] :: page_table
pagedirectory dir_name[960] :: page_table1
```

PMODE

Builds mixed-mode applications.

```
PMODE seg_list  |
      pub_sym1  [[to pub_sym2 ]  | [LENGTH number ]]
```

PMODE and ***RMODE*** allow mixed-mode applications to be built properly. Use them to change the assumed mode of segments or parts of segments.

- Use ***PMODE*** *seg_list* to mark an entire segment as protected mode.
- Any segment or segment portion marked as protected mode will reference segments using their protected-mode selectors.
- All public symbols used in this command (as code boundaries for specific purposes) must be in the same segment.

Examples

Given the following segments

```
DSEG – real-mode segment
PSEG – protected-mode segment
CODE_REAL – real-mode segment
```

with the following real-mode assembly code,

```
public prot_start
public prot_end

CODE_REAL segment eo;
            mov ax, RSEG
            xor ax, 2
prot_start:
            mov bx, PSEG
            mov cx, bx
prot_end:
            inc bx
end CODE_REAL
```

The example below causes the instructions `mov bx, PSEG` and `mov cx, bx` to have a protected-mode fixup:

```
pmode prot_start to prot_end
```

An alternate method is to use one public symbol to mark the beginning of the section and then to use the `LENGTH` keyword to specify how long it is. The following marks 10 bytes:

```
pmode prot_start length 10
```

PRINT

Prints public symbol information in the map file.

```
PRINT
```

The ***PRINT*** command tells the linker to print public symbol information in the map file.

- The name and location of each public symbol are listed module by module. Public symbols include symbols declared `PUBLIC` in assembly files, static C variables, global variables, and names of procedures from user modules and libraries.
- By default, no public symbolic information is put in the map file.

Example

```
print
```

RAM

Specifies segments to not be placed in the output file.

RAM *seg_list*

Use this command to specify segments that you do not want placed in the output file. This implies that the contents of the segments will be in RAM and contain uninitialized data.

- All segments not specified with this command will be put in the output files requested by the ***ABS86, ABS286, ABS386, HEX86, HEX386,*** and ***BINARY*** commands.

Example

Given the following segments:

```
data_seg, code_seg, stack_seg, temp
```

The following example places all but `data_seg` and `temp` in the output file:

```
ram data_seg temp
```

RMODE

Builds mixed-mode applications.

```
RMODE seg_list  |
      pub_sym1 [[to pub_sym2 ] | [LENGTH number ]]
```

RMODE and ***PMODE*** allow mixed-mode applications to be built properly. Use them to change the assumed mode of segments or parts of segments.

- Use ***RMODE*** *seg_list* to mark an entire segment as real-mode.
- Any segment or segment portion marked as real-mode will reference segments using their real-mode selectors.
- All public symbols used in this command must be in the same segment.

See also: ***PMODE*** in this chapter for an example use of ***PMODE***.

ROMBASE

Sets the base address of ROM.

```
ROMBASE seg_list  :: addressP
```

This command allows you to decrease hex-record addresses to set the base address of ROM.

- Use this command if you are burning your application into ROM and your ROM programmer doesn't allow you to set the ROM base address.

Example

```
rombase init_text :: 4000P
```

ROMMOVE

Increases hex-record addresses.

ROMMOVE *seg_list* :: *addressP*

Use this command if you want to locate records out of RAM in ROM or locate records to a higher address entirely within RAM or ROM.

Example

```
rommove init_tex :: 2000P
```

TASKGATE

Sets up task-gate descriptors.

TASKGATE *seg_list* :: *gate_options*

This command sets up task-gate descriptors and operates like the ***CALL*** commands defined in this chapter, with the following exceptions:

This command sets segments in *seg_list* as task-gate descriptors.

- The entry point must be a segment previously defined as a TSS.
- The option COUNT is not used.
- If the ***def_noentrygate*** macro is not used, the segment created must be at least 16 bytes long.
- Use the items in Table 3-4 below to form your *gate_options*. Items may be used in any order and may be repeated, separated by a space or comma.
- *DPL* defaults to zero.
- PRESENT | NOTPRESENT defaults to PRESENT.
- ENTRY defaults to the address stored at offset 0 within segment *seg_name*. You can use support macros to predefine these values in your assembly module. See the macro file **link.inc,** which is located in the directory where you installed the linker (see Chapter 1 under *Features* for a list of the macros).

Table 3-4: Gate Attributes for TASKGATE

Gate Attributes	Descriptor Correspondence
DPL=number	DPL bits
PRESENT | NOTPRESENT	Present bit
ENTRY=ptr_value	Code location the gate vectors to

Example

The assembly startup code should contain the following, which is a macro invocation:

```
; using def_noentrygate macro because entry must be a
; segment rather than a public symbol
def_noentrygate   taskx,2,0 ; in assembly invoke macro
   ; last parm isn't used
   ; with taskgate
```

The C code would look like:

```
void far taskx_gate (void);
.
.
.
procx()  {
.
.
.
taskx_gate();     // control will now switch to taskx
.
.
.
}
```

The linker command file would have the following line in it:

```
// tssx is defined as a TSS
taskgate taskx::entry=segment tssx present
```

TRAP286 and TRAP386

Set up trap-gate descriptors.

```
TRAP286 seg_list  :: gate_options
TRAP386 seg_list  :: gate_options
```

These commands set up trap-gate descriptors and operate like the ***CALL*** commands shown in this chapter, with the following exception:

- The option `COUNT` is not used.
- When you define a trap gate using ***TRAP286*** or ***TRAP386***, you must include a command to place the gate in the IDT.
- If the ***def_gate*** macro is not used, the segment created must be at least 16 bytes long.

Example

The following example creates a segment, defines it as a trap gate, then places it in slot 50 of the IDT (this assumes that the segment `idtsys` has already been created):

```
create trap_gt

trap286 trap_gt :: dpl=1 present

+ entry=init_text

idt idtsys[50]  :: trap_gt
```

TSS286 and TSS386

Set segments and specify TSS fields.

```
TSS286 seg_list  :: assign_list
TSS386 seg_list  :: assign_list
```

These commands set segments in *seg_list* as TSS segments and allow you to specify TSS fields using *assign_list*.

- 16-bit segments may be defined in your application. 32-bit segments must be created with the ***CREATE*** command.
- An initial TSS is created only for protected-mode applications, and only when one of the commands above is used. The first TSS defined in the command file is the initial TSS.
- All fields not explicitly set are left intact.
- Use the SEGMENTOF and OFFSETOF keywords to specify what part of a public symbol's address to use.

See also: *Command Syntax Elements* in this chapter for a list of command syntax elements

NOTE: TSS descriptors can only be placed in the GDT. Attempting to place them in the IDT or LDT results in an error.

Examples

The following example builds a TSS called tss_new. CS:EIP, DS, and FL.IF are explicitly set, while all other fields are left unchanged (note the use of the line-continuation character at the beginning of the second line):

```
tss386 tss_new :: cs:eip=main, ds=data_seg,
+       fl.if=0x1
```

The next example builds an initial TSS, sets the CS:EIP, and sets fields in two other TSS segments as well:

```
tss386 tss_init :: cs:eip=init_code
tss386 task_1 :: ax=2
tss386 task_2 :: efl.if=1
```

See *TSS286 Fields* and *TSS386 Fields* below for applicable TSS fields.

TSS286 Fields

AX=*number* BX=*number* CX=*number* DX=*number*

SI=*number* DI=*number* BP=*number*

DS=(*number* | *seg_name* | *pub_sym*)

ES=(*number* | *seg_name* | *pub_sym*)

CS:IP=(*number*:*number* | *seg_name* | *pub_sym*)
  CS=(*number* | *seg_name* | *pub_sym*)
  IP=*number*

SS:SP=(*number*:*number* | *seg_name* | *pub_sym*)
  SS=(*number* | *seg_name* | *pub_sym*)
  SP=*number*

SS0:SP0=(*number*:*number* | *seg_name* | *pub_sym*)
  SS0=(*number* | *seg_name* | *pub_sym*)
  SP0=*number*

SS1:SP1=(*number*:*number* | *seg_name* | *pub_sym*)
  SS1=(*number* | *seg_name* | *pub_sym*)
  SP1=*number*

SS2:SP2=(*number*:*number* | *seg_name* | *pub_sym*)
  SS2=(*number* | *seg_name* | *pub_sym*)
  SP2=*number*

LDTR=(*number* | *seg_name* | *pub_sym*)

LINK=(*number* | *seg_name* | *pub_sym*)

```
FL=number
   FL.NT=number  FL.IOPL=number
   FL.OF=number  FL.DF=number   FL.IF=number
   FL.TF=number   FL.SF=number   FL.ZF=number
   FL.AF=number  FL.PF=number   FL.CF=number
```

TSS386 Fields

```
EAX=number    EBX=number    ECX=number    EDX=number

ESI=number    EDI=number    EBP=number

DS=(number | seg_name | pub_sym)

ES=(number | seg_name | pub_sym)

FG=(number | seg_name | pub_sym)

GS=(number | seg_name | pub_sym)

CS:EIP=(number:number | seg_name | pub_sym)
         CS=(number | seg_name | pub_sym)
         EIP=number

SS:ESP=(number:number | seg_name | pub_sym)
         SS=(number | seg_name | pub_sym)
         ESP=number

SS0:ESP0=(number:number | seg_name | pub_sym)
         SS0=(number | seg_name | pub_sym)
         ESP0=number

SS1:ESP1=(number:number | seg_name | pub_sym)
         SS1=(number | seg_name | pub_sym)
         ESP1=number

SS2:ESP2=(number:number | seg_name | pub_sym)
         SS2=(number | seg_name | pub_sym)
         ESP2=number

LDTR=(number | seg_name | pub_sym)

LINK=(number | seg_name | pub_sym)
```

```
EFL=number
        EFL.NT=number   EFL.IOPL=number
         EFL.OF=number   EFL.DF=number   EFL.IF=number
         EFL.TF=number   EFL.SF=number    EFL.ZF=number
         EFL.AF=number   EFL.PF=number   EFL.CF=number
         EFL.RF=number   EFL.VM=number   EFL.CF=number

CR3=number
        CR3.PDBR=number   CR3.PCD=number
CR3.PWT=number

IO_MAP=number   TRAP=number
```

VERBOSE

Prints additional information to the map file.

```
VERBOSE  ON  |  OFF
```

You can use this command to print additional information to the map file.

- The default is ***VERBOSE OFF***.
- When ***ON***, the linker prints detailed information to the map file as each command executes. The information printed depends on the command. For example, if the command just executed impacts segment location, detailed information about where and how the segment was located is placed in the map file.
- ***VERBOSE*** can be used anywhere in the command file, and can be turned on or off as often as you wish in the same file.

Example

The following exemplifies the sort of output provided by ***VERBOSE ON:***

```
[13] verbose on

[14] create SYS_GDT
    >>>  Created segment: SYS_GDT.

[15] create raminit
    >>>  Created segment: RAMINIT.

[16] gdt SYS_GDT[1..2] :: reserve
    >>>  Defining SYS_GDT as a gdt

[17] gdt SYS_GDT::_TEXT group DGROUP _BOOT RAMINIT *
    >>>  FLAT_CODE assigned GDT[3]
    >>>  DGROUP assigned GDT[4]
    >>>  _BOOT assigned GDT[5]
    >>>  RAMINIT assigned GDT[6]
```

```
    >>>  FLAT assigned GDT[7]
    >>>  SYS_GDT assigned GDT[1]

[18] init32p raminit :: _data _bss
    >>>  Defining RAMINIT as a 32-bit protected mode RAM
init table
    >>>    _DATA placed into table.
    >>>    _BSS placed into table.
```

A. Error and Warning Messages

A

Chapter Contents

Overview

The linker generates messages when it cannot execute a command or process your application as specified.

There are three kinds of messages, organized in this chapter as follows:

1. Internal errors	Processing halts immediately—no output files are generated.
2. Errors	Processing continues—no output files are generated.
3. Warnings	Processing continues until completed—output files are generated.

Where possible, messages are listed in the following format:

1. ***** message** or **<message>**
2. Explanation that describes why the message was displayed
3. What to do to eliminate the problem here or avoid it in the future

Internal Error and Error Messages

< Internal error [- message] >

The linker has encountered either data or a situation that was thought to never occur but has in this particular case.

Please report this error to your vendor (see title page for contact information), along with as much information as possible on why this error might have occurred.

***** ERROR: <name> is not valid for <command>: <value>**

The segment <value> has been marked as a special segment of type <name>. This special segment cannot be used with <command>.

***** ERROR: <name> is <value> – Not permitted in <table>**

The selector <name> has been labeled as a special type of selector (<value>). This selector is not permitted in <table>. Refer to your processor reference manual for more information on which selectors are valid in which processor tables.

***** ERROR: <name> was already assigned to LDT[<index>]**

The given <name> was already assigned an entry in a local descriptor table.

***** ERROR: <name> was already assigned to GDT[<index>]**

The given <name> was already assigned an entry in the global descriptor table.

***** ERROR: <table> <name>[<index>] is already used**

***** ERROR: <table> <name>[<index>] is already reserved**

An attempt was made to assign a group to a <table> selector which has already been reserved.

***** ERROR: Alias segment was previously located: <name>**

Since the segment <name> has already been located, it is not possible for <name> to be an alias.

***** ERROR: Bad fixup generated in RAMINIT segment**

A fixup for the initialization table is out of range of the table.

***** ERROR: Cannot create <kind> – <name> is part of a group**

Special segments must be in their own unique selector. Therefore, they cannot be contained in a group.

***** ERROR: Cannot initialize a RAM init table: <name>**

The <name> specified is a RAM initialization segment. A packed RAM initialization segment cannot be nested inside another RAM initialization segment.

***** ERROR: Cannot place <kind> into <table>: <name>**

Certain processor structures can only contain certain special selectors. The selector <name> is not an appropriate type for <table>.

***** ERROR: Can’t alias <kind>: <name>**

The segment <name> is a special segment of type <kind> which cannot be aliased.

***** ERROR: Class <name> specified twice**

The class <name> has already been specified in the command.

***** ERROR: Command file not found: <filename>**

The linker could not find a file with the specified filename. Recheck the spelling of your command filename.

***** ERROR: Command specified twice: <value>**

The command entered has already been specified in the command file and cannot be specified more than once.

***** ERROR: COUNT option only valid for call gates**

The COUNT option is not valid with task gates, trap gates or interrupt gates.

***** ERROR: CPU not defined**

A CPU has not been defined as the target for this application. Please add the CPU command to your command file.

***** ERROR: EXCEPT specified multiple times**

You have specified the EXCEPT keyword multiple times within a command. Once you use the EXCEPT keyword, all segments, classes, and groups which follow are not processed by the command (e.g. LOCATE).

***** ERROR: Expected a '::' separator**

The linker was expecting to find a "::" separator next in parsing the command file.

***** ERROR: Expected '=' after <name>**

The linker was parsing an assignment and expected to find an '=' following <name>.

***** ERROR: Expected class name after CLASS**

The CLASS keyword was specified but the token following is not a valid class name.

***** ERROR: Expected decimal number after <token>**

The linker recognized <token> and was expecting to find a decimal number next in parsing the command file.

***** ERROR: Expected entry field name**

The linker was expecting to find the name of an entry field. Refer to Chapter 5 for a list of ENTRY fields.

***** ERROR: Expected fixup field type**

The linker was expecting to find the kind of fixup you wish to apply. Refer to Chapter 5 for a list of FIXUP kinds.

***** ERROR: Expected group name after '<value>'**

The linker expected to find the name of a group following <value>.

***** ERROR: Expected locate address after '::'**

The linker was attempting to parse an address but the value after the separator ('::') is not a valid address. An address is either a physical or linear address. A linear address has an 'L' suffix while a physical address has a 'P' suffix.

***** ERROR: Expected new segment name after '<value>'**

The name of a new segment was expected after <value>.

***** ERROR: Expected number in range 0..3 for DPL**

The number entered is not a valid number for the descriptor privilege level.

***** ERROR: Expected number in range 0..31 for COUNT**

The number entered is not in the range listed. The COUNT field in the 32-bit call gate descriptor is only five bits long.

***** ERROR: Expected ON or OFF switch after <name>**

The linker was expecting 'ON' or 'OFF' following <name>.

***** ERROR: Expected processor name after '<value>'**

The linker was expecting a processor name to follow <value>. Please refer to the syntax for the command for more information.

***** ERROR: Expecting public symbol after '<value>'**

The linker was parsing a command and was expecting to find a public symbol next in the command file.

***** ERROR: Expected segment list**

The EXCEPT keyword was given but it does not follow a list of segments, classes, or groups. Specify a list of segments, classes or groups and then use the EXCEPT keyword to list segments, classes, or groups which you don't want processed.

***** ERROR: Expected segment name after 'SEGMENT'**

The linker recognized the SEGMENT keyword and was expecting to find a segment name next in parsing the command file.

***** ERROR: Expected segment name after '<value>'**

The linker was expecting to find a segment name following <value>.

***** ERROR: Expected TSS field name**

The linker was expecting to find the name of a TSS field. See Chapter 5 for a list of valid TSS fields.

***** ERROR: Expected value after '<name>'**

The linker was expecting a value after <name>. The value could be a numeric or a symbolic such as the name of a public symbol or segment name.

***** ERROR: Expecting group name after '<value>'**

A group name was expected to follow <value>.

***** ERROR: Expecting LIMIT= or LIMIT+=**

The syntax specified for the LIMIT attribute is invalid. Use either LIMIT= or LIMIT+=.

***** ERROR: Expecting offset after <value>**

The linker was parsing a logical address and was expecting to find an offset following <value>.

***** ERROR: Expecting object filename**

The linker recognized the OBJECT keyword and was expecting to find a filename next in parsing the command file.

***** ERROR: Expecting right closing bracket**

The linker was expecting to parse a right bracket (“]”) but found something else. Review the syntax for the command.

***** ERROR: File already specified: <filename>**

A module in a library has been detected to be a <filename> which is already part of the application.

***** ERROR: <filename> already specified**

The filename you specified with one of the output commands (e.g., ABS386) has already been specified as an output filename or you are trying to use an input filename as an output filename as well.

***** ERROR: Fixup value (<value>) too large for field size (2 bytes).**
Fixup was to be applied at offset <offset> in segment <segname>.
Module is <modname>.

This will occur if the linker attempts to assign address which is too large for the field which is to hold the address. For example, 11000P is too large of an address to be the target of a 16-bit near jump.

***** ERROR: GDT[<index>] already used**

The <index> in the GDT is already in use and cannot be reserved.

***** ERROR: GDT was previously defined**

A GDT has already been created for your application. Only one GDT can be specified. The line following this message lists the line number in the command file where the GDT was previously defined.

***** ERROR: Group already located: <value>**

The group specified was located with another LOCATE command. The group cannot be located twice.

***** ERROR: Group <name> specified twice**

The group <name> has already been specified in this command.

***** ERROR: Incompatible types**

An assignment was specified between field in the TSS and a value but the types are not compatible.

***** ERROR: Incompatible types <name> = <value>**

An assignment was specified between <name> and <value> but the types are not compatible.

***** ERROR: Index out of range <low>..<high>: <value>**

The <value> specified is not within a valid range for the processor structure.

***** ERROR: Init table cannot initialize self: <name>**

The <name> specified is also the name of the initialization segment. The linker cannot pack the segment which will contain packed segments.

***** ERROR: Init table too large by <value> bytes.**
Segment <name> is a fixed size
***** ERROR: Init table too large by %lu bytes.**
Segment %s is limited to 64K

The segment for the initialization table is not large enough to hold the table.

***** ERROR: Invalid attribute option: <name>**

The <name> specified is not an option for the ATTRIBUTE command.

***** ERROR: Invalid command: <value>**

The <value> found is not recognized as a valid linker command.

***** ERROR: Invalid ENTRY field name: <value>**

The <value> specified is not a valid ENTRY field name. Refer to Chapter 5 for ENTRY field names.

***** ERROR: Invalid fixup field type: <value>**

The <value> specified is not a valid fixup kind. Refer to Chapter 5 for FIXUP kinds.

***** ERROR: Invalid gate option: <value>**

The <value> entered was not recognized as a valid gate option.

***** ERROR: Invalid initial code address**

The address specified as the starting code address has an invalid code selector or an offset which isn't within the range of a known selector. Use the ENTRY command or TSS386 command to specify the starting address for your application.

***** ERROR: Invalid number: <value>**

The <value> specified is not valid for this command. Review the syntax for the command.

***** ERROR: Invalid processor name: <name>**
Default is 80386

The <name> specified was not recognized as a valid processor. Please refer to Chapter 5 for a list of valid CPU names.

***** ERROR: Invalid range: <value1>..<value2>**

The number you specified for <value2> is smaller than <value1>. Specify the low value of the range first followed by the high value.

***** ERROR: Invalid TSS field name: <name>**

The linker was expecting to find the name of a TSS field. See Chapter 5 for a list of valid TSS fields.

***** ERROR: Location to fix (<value>) is outside module**
<module>'s contribution to segment <name>"

The module containing the public symbol does not have a range over the segment which includes the fix location.

***** ERROR: Memory allocation error**

The linker was attempting to allocate memory for processing an object file and was unable to allocate necessary memory.

***** ERROR: Module <name> symbol <value> already defined**

The public symbol has already been defined in this module. The compiler may have generated incorrect public symbol information for this module.

***** ERROR: No output specified**

The linker was ready to generate one or more output files but no command was specified for output. Specify ABS86, ABS286, ABS386, HEX86, HEX386 or BINARY to generate an output file.

***** ERROR: No room in <table> <name1>[<low>..<high>] for <name2>**

Too many segments needed to be assigned selectors in <table> than there was room for in the range specified.

***** ERROR: Object has no real-mode address: <name>**

The segment <name> has no address within the real-mode address space.

***** ERROR: RESERVE specified without range**

The linker was expecting a range to follow the RESERVE keyword. Review the syntax for the command you are trying to specify.

***** ERROR: Segment already exists: <value>**

The segment specified already exists as a segment in the application.

***** ERROR: Segment already located: <value>**

The segment specified has already been located by the linker.

***** ERROR: Segments cannot be reduced in size**

Segment limits may be increased but the linker does not allow them to be decreased.

***** ERROR: Segment <name> cannot alias itself**

Segments are not allowed alias themselves.

***** ERROR: Segment <name> specified twice**

The segment <name> has already been specified in the command.

***** ERROR: Segment <segname> is <segsize1> in module <modname>**
but is <segsize2> in module <modname>

In one module segment <segname> was defined as USE16. In another module the segment was defined as USE32. Addressing problems will result if this problem is not resolved. The same segment cannot be USE16 in one file and USE32 in another file.

***** ERROR: Task gate entry requires a selector: <value>**

The <value> found for the entry point of a task gate must be a segment previously defined as a TSS.

***** ERROR: Unexpected characters at end of command: '<value>'**

The linker thought that parsing was complete for the command but additional characters were found at the end of the command.

***** ERROR: Unknown class <name>**

The linker could not find a class with the given name.

***** ERROR: Unknown group <name>**

***** ERROR: Unknown group '<name>' after '<value>'**

The linker could not find a group with the given name.

***** ERROR: Unknown public symbol <name> after <value>**

The public symbol, <name>, was not found in the linker's list of public symbols.

***** ERROR: Unknown segment name: <name>**

***** ERROR: Unknown segment '<name>' after '<value>'**

The segment, <name>, is not a valid segment name.

***** ERROR: Unsupported object format: <filename>**

The linker did not recognize the object format found in <filename>. The linker supports object files created by Microsoft, Borland, Watcom and MetaWare tools.

***** ERROR: Use INIT16R, INIT16P, or INIT32P for INIT command**

The syntax specified is not valid for the linker.

Warning Messages

***** WARNING: <name> is undefined**

The given symbol name is undefined in the application.

***** WARNING: <message> in :<name>,**
NULL type used

***** WARNING: <message>,**
NULL type used

The linker was attempting to translate type information and found a problem given by <message>.

***** WARNING: <name> is <value> bytes long**

The given processor table (<name>) has a length of zero or the length is not a multiple of eight.

***** WARNING: Cannot open library <libpath>**

The specified library file could not be found.

***** WARNING: Constants not supported,**
:<module>.<symbol> discarded

The linker does not support constant symbols.

***** WARNING: Creating binary file over 1 Meg**

The linker was attempting to translate type information and found a problem given by <message>.

***** WARNING: Line :<name>#<value> specified multiple times**
Record ignored

When an object file defines a line number multiple times, the linker ignores the second definition.

***** WARNING: No Absolute file specified, no debug information generated**

Debug information is only stored in an absolute file. Binary and hex files contain no debug information.

***** WARNING: Segment <name1> overlaps <name2>**

The two segments are attempting to share the same memory space. Look at where you have specified the segments should be located in your command file and compare this information to the Segment Map in the map file.

***** WARNING: Segment assigned to group <group1> in module <module1>**
Reassignment to group <group2> in module <module2> ignored

The segment was found in <module1> to be assigned to <group1>. In another module, the segment was reassigned to a different group. NULL group means no group was assigned in the module. The linker will use the first assignment and ignore the second group. This warning may be a problem if the segment is a data segment.

***** WARNING: Symbol <name> assigned to register**

The symbol name has been flagged by the compiler as being assigned to a register by the optimizer. It therefore cannot be evaluated.

***** WARNING: Symbolic name <name1> too long,**
truncated to '<name2>'

The maximum length of symbol names is 160 characters.

***** WARNING: Symbols section corrupted for :<name>**

The linker has found the symbols section module <name> does not end as expected.

***** WARNING: Unknown fixup kind: <value> module = <name>**

The fixup generated by the compiler is not recognized by the linker.

***** WARNING: Unknown procedure model <value>,**
far 16-bit assumed

The model for the return value of a procedure was not recognized by the linker.

***** WARNING: Unknown register for :<module><procedure>.<name>,**
symbol discarded

The linker was attempting to processor a register variable but the register specified to contain the variable is not known.

***** WARNING: Unknown segment <name>, segment ignored**

The segment name specified is not recognized as a valid segment. Verify that it has the correct spelling.

***** WARNING: Unresolved symbol: <name>**

The symbol name was not resolved by the object files and library files specified in the command file.

***** WARNING: Unknown symbols format in <name>, data ignored**

The linker did not recognize the format of symbols in module <name>.

***** WARNING: Unknown types format in <name>, data ignored**

The linker has detected a type definition record in module <name> which it does not recognize.

***** WARNING: Unsupported language encountered: <value>**

The linker read the compiler record in the symbol information and the value read was not recognized by the linker.

***** WARNING: Unsupported symbol encountered: <value>**

The linker has detected a symbol which it does not support. The <value> is the symbolic record type.

***** WARNING: 'With Start' record encountered in :<name>**

The linker has detected a symbolic record which it does not support.

This page contains only this line of text.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z

I